ProverSLD: An Interactive Visualizer for Teaching SLD Resolution in Prolog and First-Order Logic

Ján Perháč*, Patrik Čmil and Samuel Novotný

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic

Abstract

We present ProverSLD, an interactive web-based tool for teaching and visualizing the SLD resolution process. The system supports first-order logic and a fragment of Prolog syntax. It provides a complete transformation pipeline from formulas to clausal form, visualizes the full SLD tree, and enables synchronized, step-by-step derivation via a resolution table linked to the tree. Users can choose between depth-first and breadth-first search strategies. We conducted a user study where the tool was deployed and evaluated in classroom settings, yielding positive feedback and results.

Keywords

Interactive visualization, LaTeX code generation Logic programming, Prolog, SLD resolution, Teaching tool,

1. Introduction

Many computer science programs include courses on logic programming as part of their curriculum. However, this paradigm is often introduced as an "exotic" style of programming. Students typically learn to write simple programs through specific examples, but they are rarely given a deeper understanding of what happens behind the scenes. In particular, they are often not taught how computation is actually performed—namely, that it relies on Selective Linear Definite clause resolution (*SLD Resolution*), a theorem-proving process used to derive logical conclusions.

This lack of insight poses challenges for students to grasp more advanced concepts, such as the cut operator in Prolog. Without a solid understanding of the underlying logical proof process, the behavior and implications of such constructs can seem unclear or arbitrary.

Prolog, the most widely known logic programming language, is based directly on SLD resolution. SLD resolution is a specialization of the resolution method, which dates back to the early work by Davis and Putnam [1] and was later formalized by Robinson [2] for definite clauses. In practice, understanding SLD resolution is challenging for several reasons. Before the derivation can begin, the input formula must be transformed into clausal form through multiple error-prone steps, where a single mistake can lead to an incorrect result. Unification requires careful tracking of variable substitutions throughout the derivation, as substitutions from earlier steps propagate to subsequent ones. For non-trivial examples, the SLD tree may contain many branches, making manual construction impractical. These challenges motivate the need for an interactive visualization tool to support students throughout the entire process.

Existing tools only partially address this problem. Some applications focus on exporting the final SLD tree [3], while others cover only the conversion of formulas into clausal form [4]. Key features such as step-by-step visualization of the SLD tree, support for both first-order logic and Prolog syntax, and the ability to choose a proof tree search strategy are often missing or only partially implemented. Our analysis suggests that no single tool currently provides a complete workflow, from entering a logical formula or a Prolog program to an interactive visualization of the full SLD tree.

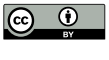
PEG 2026: Fourth Prolog Education Workshop, July 19, 2026, Lisbon, Portugal

*Corresponding author.

jan.perhac@tuke.sk (J. Perháč); patrik.cmil@student.tuke.sk (P. Čmil); samuel.novotny@tuke.sk (S. Novotný)

https://kpi.fei.tuke.sk/en/person/jan-perhac (J. Perháč); www.linkedin.com/in/patrik-ÄĽmil-a08b58334 (P. Čmil); https://kpi.fei.tuke.sk/en/user/11125 (S. Novotný)

0000-0001-6347-2409 (J. Perháč); 0009-0004-0995-1459 (P. Čmil); 0009-0005-1675-2965 (S. Novotný)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-03

CEUR-WS.org/Vol-4311/PEG_regular_06.pdf

In this paper, we present ProverSLD[1] (*available online* [5]), an interactive web-based tool designed as a teaching aid. The tool provides a visualization of the SLD tree, with color-coded successful and failed branches. Users can navigate through the SLD tree step by step, while the corresponding resolution process is displayed in tabular form. The table is linked to the tree and shows the corresponding derivation at each step. The tool also allows users to choose between depth-first and breadth-first search and to generate LaTeX code for both the SLD tree and the resolution table. ProverSLD is intended for student self-study as well as for use by instructors as a demonstration tool during lectures.

The tool's usability was evaluated in a real classroom setting through a user study involving students enrolled in a master's-level logic course at the Technical University of Košice, yielding positive results.

This paper is structured as follows. Section 2 presents the tool's architecture, design, and functionalities, along with motivating examples. Section 3 describes the classroom-based user study and its evaluation. Section 4 discusses related work, and Section 5 concludes the paper.

2. The Tool

This section presents the design and implementation of the ProverSLD tool. We describe the technologies used, the input processing pipeline for first-order logic and Prolog syntax, and the main functionalities, including formula transformation, SLD tree visualization, and resolution table interaction.

2.1. Technologies Used

Modern web applications are often implemented as single-page applications (SPAs), which are loaded only once and subsequently update their content dynamically without reloading the page. This approach improves user experience but imposes higher demands on application design.

The most commonly used frontend technologies include React, Vue, and Angular. React is a library based on a component-oriented approach and a virtual Document Object Model (DOM), enabling efficient User Interface (UI) updates and good code organization. Vue emphasizes simplicity and readability but has a smaller ecosystem. Angular is a comprehensive framework with built-in tools, suitable mainly for large-scale applications, but it introduces higher complexity.

For the development of our application, we chose React, primarily due to its extensive ecosystem and strong support for visualization. For visualizing the SLD tree, we considered D3.js and React Flow. D3.js offers high flexibility but requires more manual implementation and integrates less smoothly with React. In contrast, React Flow provides native integration, built-in support for interactive graph structures, and, when combined with the Dagre library, automatic node layout. For these reasons, we selected React Flow, as it enables simpler implementation and clear visualization of tree structures.

2.2. Parsers and languages

Input processing in the application consists of tokenization and parsing. For tokenization, we implemented a custom lexer, which provides precise control over input processing and enables high-quality error reporting, while avoiding the unnecessary complexity of lexer generators. For parsing, we adopted a recursive descent approach and implemented two dedicated parsers: one for first-order logic and one for a fragment of Prolog syntax.

After tokenization, the application determines the type of input. We implemented a function that scans the token sequence for Prolog-specific elements such as facts (.), rules (:-), and queries (?-). If any of these are detected, the input is passed to the Prolog parser; otherwise, it is processed by the first-order logic parser. This approach enables automatic detection of the input type without requiring user intervention, simplifies the user experience, and ensures that the appropriate parsing strategy is applied transparently and consistently. The process is illustrated in Figure 1.

[1] Available at https://proversld.vercel.app/

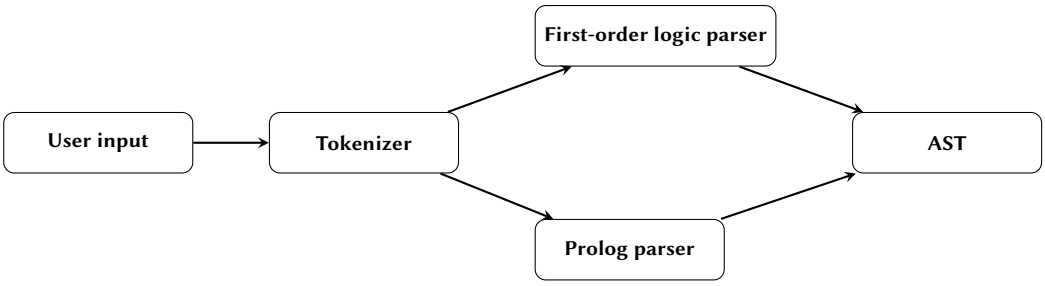


Figure 1: Input parsing pipeline

This design allows us to handle the differences between the two languages explicitly, resulting in clearer structure, better maintainability, and more precise and user-friendly error messages. At the same time, it provides full control over the parsing process and ensures flexibility for future extensions.

2.2.1. First-order logic parser

The parser supports the following language of first-order logic formulas described in Backus–Naur Form (BNF):

$$A ::= P(t, \ldots, t) \mid P \mid A \wedge A \mid A \vee A \mid A \Rightarrow A \mid \neg A \mid (\forall x)A \mid (\exists x)A,$$

where P denotes a predicate without arguments (a proposition), and $P(t, \ldots, t)$ denotes a predicate with arguments. The logical connectives include conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and negation ($\neg$). The symbols $\forall$ and $\exists$ represent the universal and existential quantifiers, respectively.

Terms are defined as follows:

$$t ::= x \mid c \mid f(t, \ldots, t),$$

where x is a variable that must be bound by a quantifier, c is a constant, and f is a function symbol. Additionally, the input formula must contain at least one binary connective ($\Rightarrow$, $\vee$, $\wedge$).

Each operator has a defined precedence and associativity that determines the order of evaluation in expressions without parentheses. The operator precedence and associativity are summarized in Table 1.

Table 1
Precedence and associativity of logical operators in first-order logic syntax

Precedence	Associativity	Symbol
1 (lowest)	Right	$\Rightarrow$
2	Right	$\vee$
3	Right	$\wedge$
4	Right	$\neg$
5 (highest)	Right	$\forall, \exists$

Although conjunction ($\wedge$) and disjunction ($\vee$) are associative operators, a specific associativity must be chosen for implementation purposes. This ensures a unique structure of the abstract syntax tree during parsing, even though the semantic meaning of the formula remains unchanged. In our implementation, we adopt right-associativity for these operators.

The parser does not consider if identifiers are uppercase and lowercase and relies solely on the structure of the formula. The rules for distinguishing identifier types are summarized in Table 2.

An identifier is considered a variable if it is bound by a quantifier ($\forall$ or $\exists$). An identifier that is not bound by any quantifier and is not followed by parentheses is treated as a constant. Numeric values are also considered constants. If an identifier is immediately followed by an opening parenthesis, its type depends on its position in the formula:

- If it appears at the top level of the formula, it is treated as a predicate.
- If it is nested within the arguments of another predicate or function, it is treated as a function.

Table 2
Distinction of identifier types in first-order logic syntax

Identifier	First-order logic syntax
Variable	bound by a quantifier (∀, ∃)
Constant	not bound by a quantifier, without parentheses
Predicate	followed by parentheses, at the top level of the formula
Function	followed by parentheses, nested within arguments

2.2.2. Prolog Parser

The Prolog parser processes programs composed of three types of clauses:

- **Fact** – a predicate terminated by a period.
- **Rule** – of the form `head :- body`, where the body is a list of literals separated by commas.
- **Goal** – begins with the symbol `?-` followed by a list of literals separated by commas.

In Prolog syntax, identifier types are distinguished by the case of the first letter, as summarized in Table 3.

Table 3
Distinction of identifier types in Prolog syntax

Identifier	Prolog Syntax
Variable	begins with an uppercase letter
Constant	begins with a lowercase letter, without arguments
Predicate	begins with a lowercase letter, followed by parentheses, at the top level of a clause
Function	begins with a lowercase letter, followed by parentheses, nested within arguments

Identifiers starting with an uppercase letter are always treated as variables. Identifiers starting with a lowercase letter without arguments are considered constants, while those with arguments are interpreted as predicates or functions. Numeric values are also treated as constants.

Additionally, the Prolog parser internally transforms rules of the form `head :- body` into the equivalent implication $body \Rightarrow head$, ensuring a uniform internal representation and consistent processing regardless of the input syntax.

Currently, the parser does not support certain Prolog features such as the cut operator, negation as failure, lists, or tuples. These features are planned to be incorporated in future work.

The currently supported subset of Prolog can be described in BNF as follows:

$$
\begin{aligned}
A &::= Program\ Goal \\
Program &::= \epsilon \mid Clause\ Program \\
Clause &::= Fact \mid Rule \\
Fact &::= p. \\
Rule &::= p\ :-\ Body. \\
Goal &::= ?-\ Body. \\
Body &::= p \mid p,\ Body \\
p &::= name \mid name(t, \ldots, t) \\
t &::= X \mid a \mid f(t, \ldots, t)
\end{aligned}
$$

The program consists of zero or more facts and rules, followed by exactly one goal. X denotes a variable (starting with an uppercase letter), a a constant (starting with a lowercase letter or a number), and $name$, f denote predicate and function symbols, respectively.

Note. After parsing the input into an abstract syntax tree (AST), the subsequent processes of transforming formulas into clausal form and further processing are not described in detail due to space

limitations. These implementation details are less important in the context of the primary focus of this paper compared to the parsing stage, which defines the supported languages and their syntax. Therefore, the next part of the section focuses on the functionalities of the system. For further details, the source code of the tool is freely available in a GitHub repository [6].

2.3. Functionalities

The application runs directly in a web browser and does not require user registration. The initial screen, shown in Figure 2, consists of a navigation bar containing the `ProverSLD` logo, a `Documentation` tab, and a language switcher.

The main interface includes a text editor for entering a formula or a logic program. Above the editor, a dropdown menu provides a set of predefined examples. Below the editor, a set of buttons allows quick insertion of logical symbols, including conjunction, disjunction, implication, negation, and universal and existential quantifiers. A `Process Formula` button located in the bottom-right corner initiates the computation.

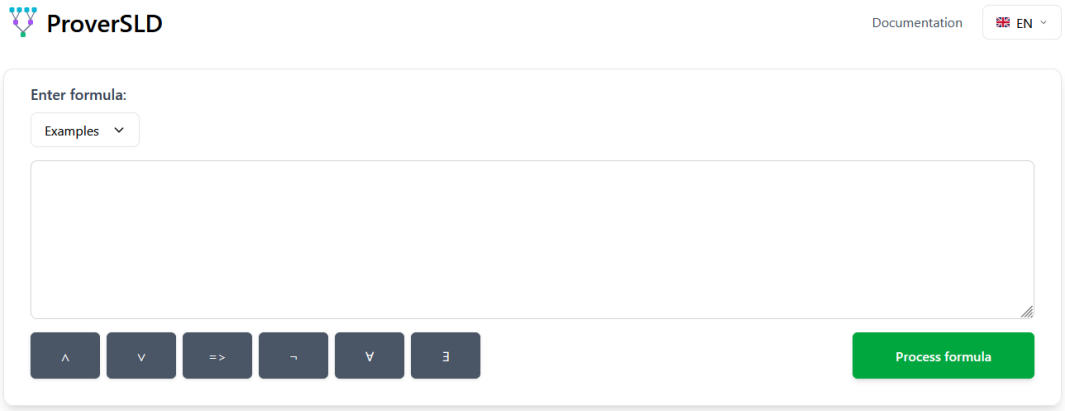


Figure 2: Initial screen of `ProverSLD`

2.3.1. Input

The application supports two input formats. In first-order logic syntax, the user enters a formula to be proved. In Prolog syntax, the user enters a logic program consisting of facts and rules together with a goal query. Each format is handled by its own parser and the input format is detected automatically without any manual configuration. Logical symbols can be inserted using the buttons below the editor or typed directly using keyboard shortcuts, as shown in Table 4.

Table 4
Supported logical shortcuts and their corresponding symbols

Shortcuts	Symbol
`\and`, `\land`, `\&`	∧
`\or`, `\lor`, `|`	∨
`\implies`, `\rightarrow`, `\to`	⇒
`\neg`, `\not`, `\!`	¬
`\forall`	∀
`\exists`	∃

2.3.2. Error messages

If the input contains a syntax error, the application displays a clear error message directly below the editor, as illustrated in Figure 3. Once the input is corrected and `Process formula` is pressed again, the error message disappears and the application proceeds with the full computation. This is followed by the transformation steps showing the conversion of the formula to clausal form, the interactive SLD tree and the resolution table showing the chronological progress of the derivation.

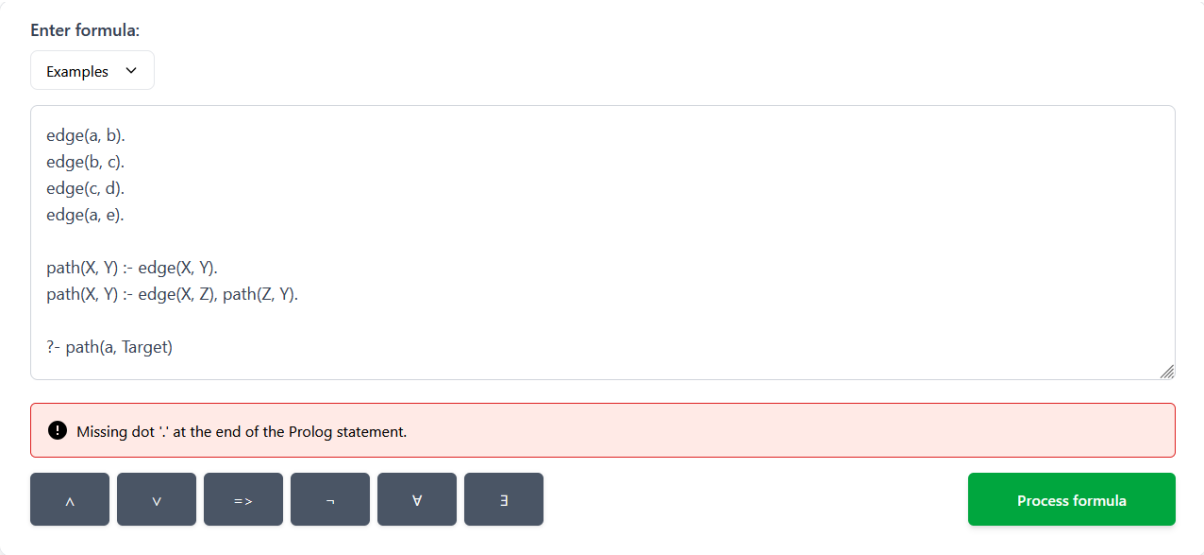


Figure 3: Error message for invalid input

2.3.3. Formula Transformation

After successful processing, the application displays the formula transformation steps. Since SLD resolution operates on formulas in clausal form, the input formula must first be converted into this form. The following steps are displayed:

- Parsed formula
- Negated formula
- Elimination of implications
- Negative normal form (NNF)
- NNF with unique bound variables
- Prenex normal form (PNF)
- Skolem normal form (SNF)
- Omission of universal quantifiers
- Conjunctive normal form (CNF)
- Set of clauses

This allows the student to follow the entire transformation process and identify any potential mistake. Figure 4 shows the transformation steps for the formula

$$(\forall x)(\forall y)(P(x) \land P(y)) \Rightarrow (\forall z)P(z).$$

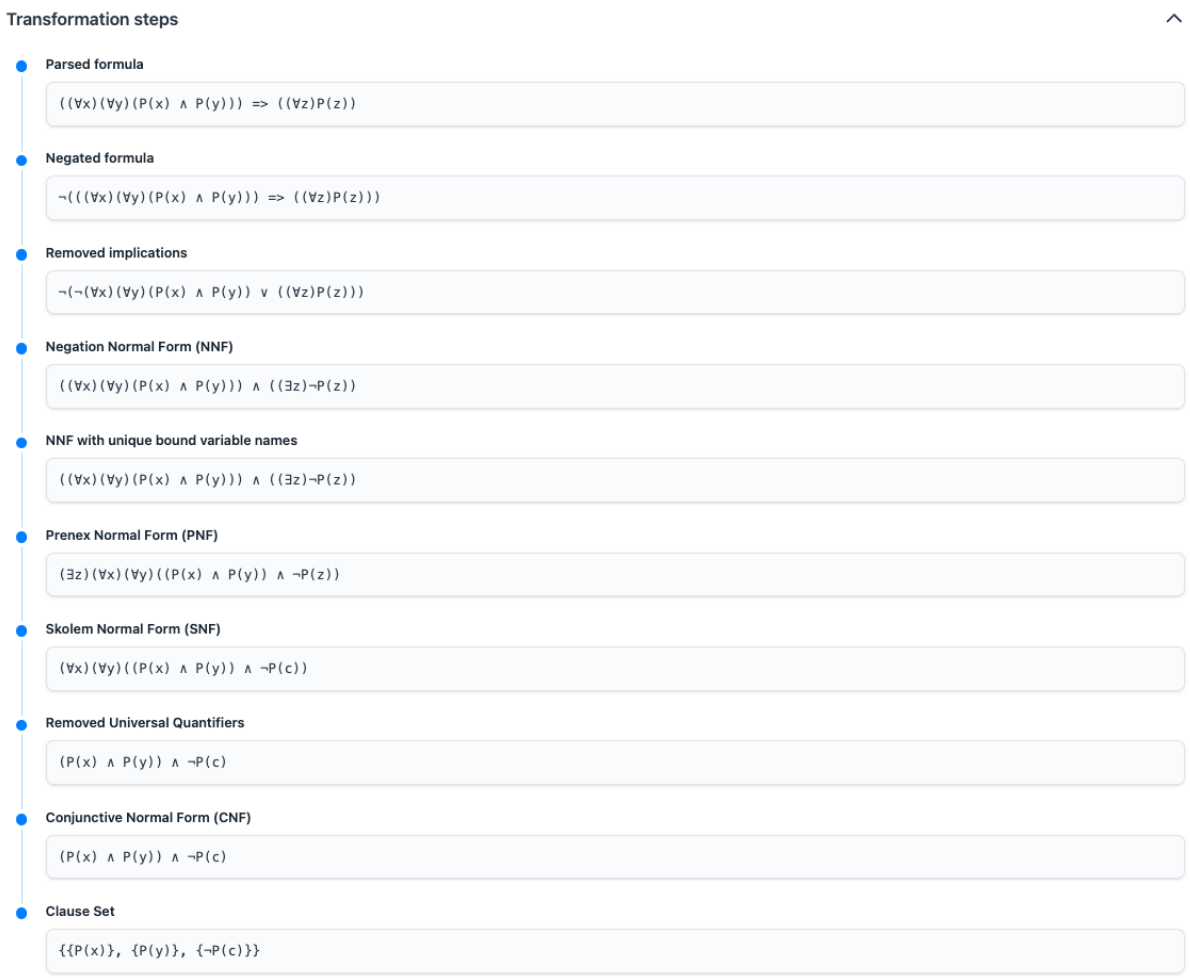


Figure 4: Transformation steps of formula $(\forall x)(\forall y)(P(x) \wedge P(y)) \Rightarrow (\forall z)P(z)$.

2.3.4. SLD Tree visualization

After the formula transformation, the application generates and displays the SLD tree as an interactive visualization. The root node represents the negated goal of the derivation. Each edge of the tree is labelled with the substitution applied in the corresponding derivation step. Nodes are color-coded according to their state. To prevent infinite loops caused by recursive rules without a termination condition, the search is automatically stopped when the tree reaches a maximum depth of 16 levels.

Let us consider the following Prolog program:

```
parent(dobroslav, dragomir).
parent(vlastislava, dragomir).
parent(dragomir, vesna).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

?- grandparent(Grandparent, Grandchild).
```

Figure 5 shows the generated SLD tree, where green nodes indicate a successful end of the derivation, and the red node indicates a failed branch. The user can zoom in, zoom out, and navigate the tree using the mouse. Above the tree, a toolbar provides controls for choosing the search strategy, step-by-step navigation, and generating LaTeX code for the tree.

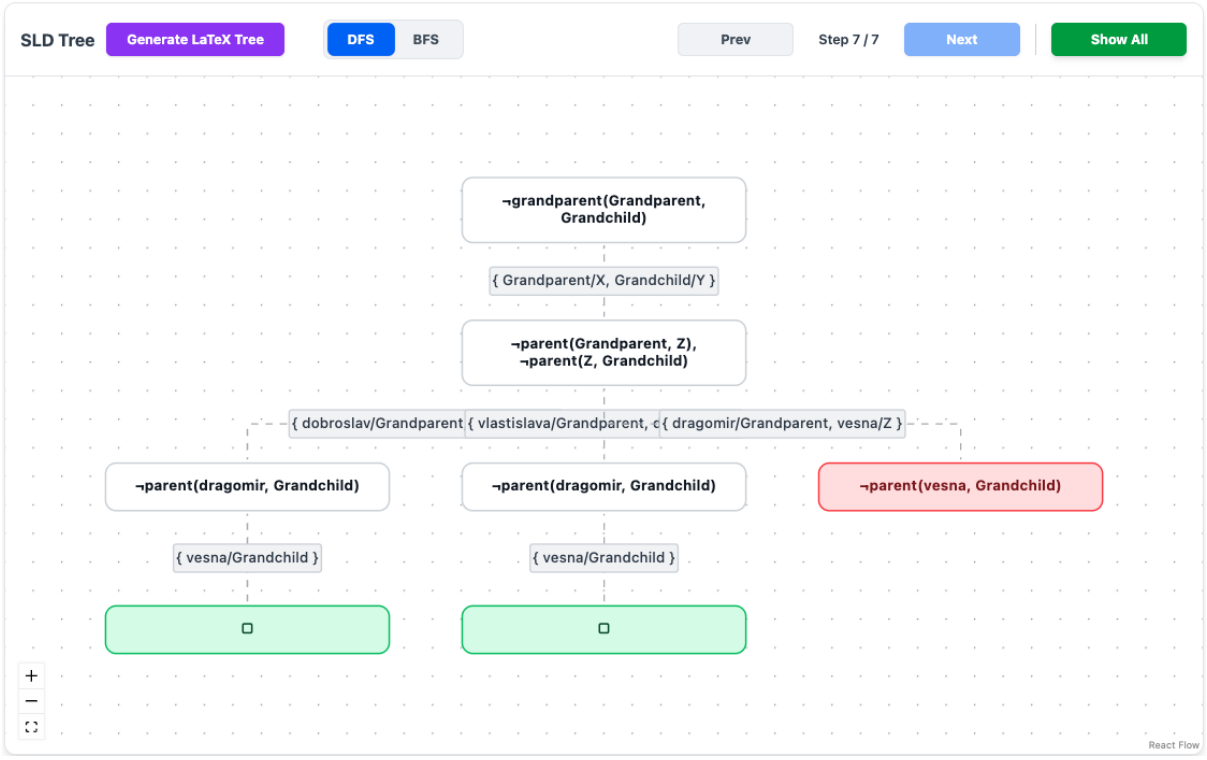


Figure 5: SLD tree visualization

2.3.5. Resolution Table

Alongside the SLD tree, the application displays a resolution table that captures the chronological progress of the derivation. At the beginning, the table contains the goal query, followed by the clauses of the knowledge base. The next rows represent the newly derived goals produced during the resolution process. The table contains the following columns:

- step number
- current goal
- references to the clauses used in the resolution step
- applied substitution

The resolution table is linked to the SLD tree. Clicking on a node in the tree highlights the corresponding row in the table and at the same time highlights the edge above the selected node. Equally, clicking on a row in the table highlights the corresponding node in the tree. The empty goal in the table indicates a successful end of the derivation, as shown in Figure 6.

2.3.6. LaTeX Code Generation

The application allows the user to generate LaTeX code for both the SLD tree and the resolution table. After clicking the corresponding button in the toolbar, a dialog is displayed with the following options:

- **Scope** — tree or table environment only, or a complete compilable document
- **Orientation** — portrait or landscape

After confirming the selection, the generated code is copied to the clipboard and can be directly pasted into a LaTeX document.

Resolution Trace [Generate LaTeX Table]

Step	Clause	Resolved	Unification
1	¬grandparent(Grandparent,Grandchild)		
2	parent(dobroslav,dragomir)		
3	parent(vlastislava,dragomir)		
4	parent(dragomir,vesna)		
5	¬parent(X,Z), ¬parent(Z,Y), grandparent(X,Y)		
6	¬parent(Grandparent, Z), ¬parent(Z, Grandchild)	1,5	{ Grandparent/X, Grandchild/Y }
7	¬parent(dragomir, Grandchild)	6,2	{ dobroslav/Grandparent, dragomir/Z }
8	□	7,4	{ vesna/Grandchild }
9	¬parent(dragomir, Grandchild)	6,3	{ vlastislava/Grandparent, dragomir/Z }
10	□	9,4	{ vesna/Grandchild }
11	¬parent(vesna, Grandchild)	6,4	{ dragomir/Grandparent, vesna/Z }

Figure 6: Resolution table linked to the SLD tree

3. User Study and Tool Evaluation

For the evaluation, we deployed `ProverSLD` in the course *Logic for Informaticians* at the Technical University of Košice during the spring semester of the 2025/2026 academic year. It is an optional master's-level course with 37 enrolled students. The course covers formal logic and its applications in informatics, as well as an introduction to logic programming, with SLD resolution as one of its key topics.

3.1. Experiment description

During the introductory lecture on logic programming, we used the tool as a visualization aid. We demonstrated its functionalities using specific examples in both first-order logic and Prolog syntax. Students were guided through the entire derivation process step by step. This included demonstrating the transformation of formulas into clausal form, showing how unification is performed at each step, observing how individual branches of the SLD tree are expanded, and identifying which branches succeed and which fail. The resolution table was used to illustrate the substitutions applied at each step and to illustrate how they propagate to subsequent goals.

To evaluate the tool, students were asked to complete two predefined user scenarios:

- The first scenario focused on the core functionality of the tool. Students entered a logic formula or Prolog program and examined the transformation steps leading to clausal form. They then expanded the SLD tree step by step, verified the color-coding of nodes, and confirmed the connection between the tree and the resolution table by interacting with individual nodes.
- The second scenario focused on error handling. Students entered syntactically incorrect input and verified that a clear error message was displayed. After correcting the input and pressing `Process Formula` again, they confirmed that the error message disappeared and the results were displayed correctly.

The usability of ProverSLD was evaluated through a user study. After completing the tasks, students were asked to fill out a questionnaire. In total, 19 students participated in the study, which took place in March and April 2026. Each participant completed the two predefined tasks and subsequently filled in the questionnaire.

3.2. Usability Evaluation

The first part of the questionnaire was dedicated to evaluating the usability of the tool. We employed the System Usability Scale (SUS) [7] as a standardized method for quantitative assessment. The SUS consists of 10 questions rated on a five-point Likert scale, producing a score in the range 0–100 that reflects the overall usability of the system.

ProverSLD achieved an average SUS score of **80.53**, with a median of 85.0 and a trimmed mean of 82.50. According to the standard adjective rating scale [8], this places the tool in the *excellent* category, corresponding to approximately the top 10% of systems in terms of usability.

3.3. Qualitative evaluation

The second part of the questionnaire focused on the qualitative evaluation of the tool. It consisted of several open-ended questions related to the completion of the scenarios, identification of encountered issues, assessment of the tool's strengths and weaknesses, and suggestions for further improvement.

3.3.1. Scenario Completion and Encountered Issues

The responses indicate that the majority of participants did not encounter any technical or usability issues while completing either of the scenarios. Most students reported that both scenarios proceeded smoothly and that the system behaved as expected, including correct error detection and handling in the second scenario.

A small number of minor issues were identified. In Scenario 1, some participants reported difficulties with navigation in the SLD tree, particularly when attempting to return to a previous step using the *Previous step* button, which occasionally did not behave as expected. Additionally, several students mentioned confusion related to the *Export* functionality, noting that the button was not immediately visible or that its naming was misleading, as it performs copying rather than downloading.

In Scenario 2, the system generally performed reliably; however, one participant suggested improving error reporting by highlighting the exact location of the error in the input.

3.3.2. Positive Feedback

The qualitative feedback highlights several strengths of the tool. The most frequently appreciated aspect was the clear and intuitive visualization of the SLD tree, particularly the use of color-coding and the interactive connection between the tree and the resolution table. Many participants also emphasized the simplicity and ease of use of the interface, noting that the tool is intuitive and effectively supports understanding of the resolution process. Additional positive aspects included step-by-step visualization of transformations, interactive exploration of the derivation, and convenient insertion of logical symbols.

3.3.3. Suggestions for Improvement

Participants also provided several suggestions for improvement. A recurring theme was enhancing the user interface, including the addition of a dark mode, clearer labeling of certain controls (e.g., the *Export* functionality), and improved visibility of some UI elements such as dropdown menus. Several users suggested extending the export functionality (e.g., allowing file download) and improving feedback mechanisms, such as highlighting errors more precisely or visually distinguishing parentheses in complex formulas. Other suggestions included adding tooltips or legends, improving navigation within the SLD tree, and refining certain interaction details.

3.3.4. Improvements Made

Based on the collected feedback, several improvements were implemented in the application. The LaTeX code generation buttons were updated to include visible text labels, improving their clarity and visibility. All interactive elements were enhanced with a pointer cursor on hover to provide clearer interaction feedback. Additionally, visual highlighting of the active unification edge was introduced when selecting a node in the SLD tree, further improving the usability and clarity of the visualization.

4. Related Work

A number of tools have been proposed to support the teaching of Prolog and logic programming with the goal of visualizing SLD resolution trees. However, most of them address only isolated aspects of the problem and fall short of providing a complex learning environment.

One of the earliest systems, introduced by Tamir et al. [9], extends a Prolog interpreter with the ability to export and visualize SLD resolution trees. While conceptually valuable, the tool reflects the technological limitations of early 1990s desktop applications and is no longer practical for contemporary use, particularly in web-based educational settings.

The approach described in [3] focuses on generating SLD resolution trees as LaTeX code, now available as the `sldnfdraw` package in SWI-Prolog. Although suitable for producing static documentation, this solution lacks interactivity, which limits its usefulness for teaching and exploratory learning. In contrast to our tool, it does not allow users to manipulate or incrementally construct derivations.

The web-based application `FolSolver` [4] supports the transformation of first-order logic formulas into clausal form and provides an interactive resolution table. However, its scope is restricted to resolution-based reasoning and does not extend to SLD tree construction.

SWISH [10] offers a modern, browser-based Prolog environment with rich features for program execution and sharing. Despite its strengths, it does not provide any native support for visualising SLD trees.

The Prolog Visualizer [11] is explicitly designed for illustrating SLD resolution. It enables step-by-step query execution. However, its representation is limited to a linear textual format that only reflects the current computation state and explored leaves. Compared to our tool, it restricts the user's ability to explore the global structure of the search tree and alternative computation paths at the same time.

Tau Prolog [12] comes closest to our objectives. As a JavaScript-based interpreter, it includes a basic derivation tree view and runs entirely in the browser. Nevertheless, its visualization capabilities remain limited, lacking a step-by-step navigation and offering no support for input in full first-order logic or LaTeX export.

In summary, existing tools are either outdated, non-interactive, narrowly focused, or lack suitable visualization capabilities. None of the reviewed tools combines all of the following functionalities:

- input in both first-order logic and Prolog syntax,
- interactive, step-by-step construction and exploration of SLD trees,
- and export of both SLD trees and resolution tables into LaTeX.

5. Conclusion

In this paper, we presented `ProverSLD`, an interactive web-based tool designed to support the teaching and learning of SLD resolution. The tool provides a complete workflow from input processing in both first-order logic and Prolog syntax, through transformation to clausal form, to interactive visualization of the SLD tree and the corresponding resolution process.

The results of the user study indicate that the tool is well-received by students and achieves a high level of usability, as reflected by the SUS score in the *excellent* range. Qualitative feedback further confirms that the visualization and interactivity contribute to improved comprehension of the resolution process.

The identified issues were minor and mainly related to user interface clarity, and were subsequently addressed in the improved version of the tool.

Future work will focus on extending the supported subset of Prolog with additional language features, such as the cut operator, negation as failure, and support for lists and tuples. Furthermore, we plan to enhance the user interface based on the collected feedback and to conduct a more extensive evaluation involving a larger group of students and long-term usage in teaching.

Acknowledgments

This research was supported by the Cultural and Educational Grant Agency (KEGA) under project No. 052TUKE-4/2025, "Modern Approaches in Educating IT Professionals in the Field of Type Theory".

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT (GPT-4) and Grammarly for grammar and spelling checking. After using these tools, the authors carefully reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the ACM (JACM) 7 (1960) 201–215.

[2] J. A. Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM 12 (1965) 23–41. doi:10.1145/321250.321253.

[3] M. Gavanelli, Sldnf-draw: Visualization of prolog operational semantics in latex, Intelligenza Artificiale 11 (2017) 81–92.

[4] B. Bekiaris, J. Perháč, Interactive proof assistant based on resolution method in first order logic, in: 2024 IEEE 28th International Conference on Intelligent Engineering Systems (INES), 2024, pp. 000167–000174. doi:10.1109/INES63318.2024.10629140.

[5] ProverSLD - web application, 2026. URL: https://proversld.vercel.app/, (accessed Apr. 28, 2026).

[6] ProverSLD — github project repository, 2026. URL: https://github.com/KirrTap/prover-sld, (accessed Apr. 28, 2026).

[7] J. Brooke, SUS: A quick and dirty usability scale, in: P. W. Jordan, B. Thomas, B. A. Weerdmeester, I. L. McClelland (Eds.), Usability Evaluation in Industry, Taylor & Francis, London, 1996, pp. 189–194.

[8] A. Bangor, P. Kortum, J. Miller, Determining what individual SUS scores mean: Adding an adjective rating scale, Journal of Usability Studies 4 (2009) 114–123.

[9] D. E. Tamir, R. Ananthakrishnan, A. Kandel, A visual debugger for pure prolog, Information Sciences-Applications 3 (1995) 127–147.

[10] J. Wielemaker, T. Lager, F. Riguzzi, SWISH: SWI-Prolog for sharing, arXiv preprint arXiv:1511.00915 (2015).

[11] Z. Lai, A. Warth, Prolog visualizer, https://cdglabs.github.io/prolog-viz, 2014. Accessed: 2026-05-07.

[12] J. A. Riaza, Tau Prolog: A Prolog interpreter for the Web, Theory and Practice of Logic Programming 23 (2023) 783–798.