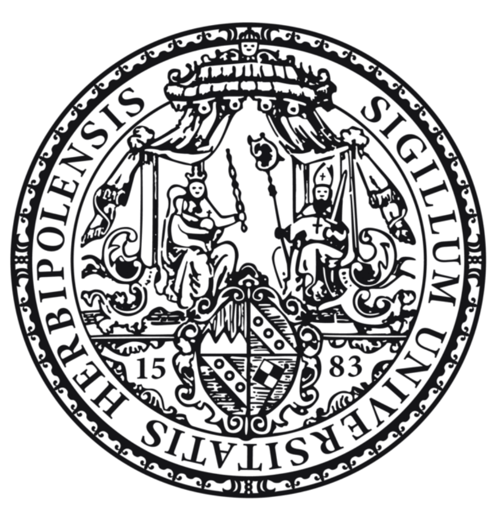
An Interactive Visualisation for Definite Clause Grammars

Master Thesis

Jona Kalkus



Supervisors:	Prof. Dr. Dietmar Seipel M.Sc. Falco Nogatz
Chair:	Chair for Computer Science I, (Algorithms, Complexity and Knowledge-Based Systems)
Institute:	Institute of Computer Science
University:	Julius-Maximilians-Universität Würzburg
Submission Date:	November 9, 2017

Zusammenfassung

Definite Clause Grammars (DCGs) stellen eine komfortable Möglichkeit zur Beschreibung von Listen in Prolog dar. In dieser Arbeit wird eine Anwendung vorgestellt, die Anfragen an diese Grammatiken interaktiv visualisiert.

Hierzu werden zunächst Informationen über die Ausführung von Anfragen benötigt. Mehrere Möglichkeiten um diese zu erhalten werden diskutiert. Der realisierte Ansatz nutzt eine modifizierte Expansion von Grammatikregeln, Anfragen an diese werden anschließend von einem Meta-Interpreter ausgewertet. Dabei werden Informationen über Ausführungsschritte gespeichert. Diese werden danach zu einem SWI-Prolog 7 Dict zusammengefasst, welches erfolgte Schritte strukturiert abbildet.

Mithilfe von Pengines, einer Bibliothek die entfernte Zugriffe auf SWI-Prolog über das Internet ermöglicht, wurde eine Webanwendung erstellt. Diese verwendet die vom Meta-Interpreter bereitgestellten Daten, visualisiert sie und bietet dem Nutzer somit die Möglichkeit interaktiv nachzuvollziehen, welche Ausführungsschritte zur Berechnung eines Ergebnisses durchgeführt wurden. Die Darstellungsweise basiert auf Parse Trees, erweitert um die Visualisierung von Fehlschlagen und Backtracking. Die Anwendung ist sowohl zum Lernen der zugrundeliegenden Konzepte, als auch zum Debuggen von Grammatiken geeignet.

Abstract

Definite Clause Grammars (DCGs) are a convenient way to describe lists in Prolog. An application which interactively visualises queries on these grammars is presented in this thesis.

First of all, information on the execution of queries is required. Several possibilities to obtain this are discussed. The realised approach uses a modified expansion of grammar rules. Queries on these are then evaluated by a meta-interpreter which traces the execution and asserts information about each execution step. These steps are then summarised into a SWI-Prolog 7 dict which represents them in a structured manner.

A web application based on Pengines, a library that allows remote access to SWI-Prolog over the internet, was implemented. This makes use of the meta-interpreter, visualises traced steps and thus offers the user the opportunity to interactively understand which execution steps were performed to calculate a result. The presentation is based on parse trees, enhanced by the visualisation of failure and backtracking. The application is suitable both for learning the underlying concepts and for debugging grammars.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Structure of this Work . . . 2

2 Related Work **3**
2.1 Language Analysis in Prolog . . . 3
2.2 Visualising the Execution of Prolog . . . 3
2.3 Interactive Visualisations for Other Programming Languages . . . 6

3 User Scenario **7**
3.1 Target User Groups . . . 7
3.1.1 Beginners . . . 7
3.1.2 Experienced users . . . 8
3.2 Application Mock-up . . . 8
3.2.1 Enter Grammars and Run Queries . . . 8
3.2.2 General Idea of Visualising the Result . . . 9
3.2.3 Integration of Additional Information . . . 10
3.2.4 Visualising Execution Order . . . 11
3.3 Summary . . . 12

4 Technology: Prolog **13**
4.1 Origins of Prolog . . . 13
4.2 Definite Clause Grammars . . . 14
4.2.1 Introductory Example . . . 14
4.2.2 Syntax of DCGs . . . 17
4.2.3 DCG Expansion in SWI-Prolog . . . 18
4.2.4 Querying DCGs . . . 20
4.3 Tracing in SWI-Prolog . . . 21
4.3.1 Execution Model . . . 21
4.3.2 Command Line Tracing . . . 22
4.3.3 Intercepting the Tracer . . . 24
4.4 Dicts . . . 25
4.5 Pengines . . . 27
4.5.1 Usage of Pengines . . . 27
4.5.2 Built-In Security Features . . . 29

5 Approaches to Analyse the DCG Execution **31**
5.1 Generating Parse Trees with DCGs 31
5.2 Using other Programming Languages than Prolog 33
5.3 Using the Built-In Tracer 34
5.3.1 Customising the Tracer 34
5.3.2 Problems Encountered 36
5.4 Using Meta-Interpreters 37
5.4.1 Vanilla Meta-Interpreter 37
5.4.2 Generating a Parse Tree 38
5.4.3 Picture Backtracking 40
5.4.4 Tracing Meta-Interpreter 40

6 DCG Tracer **45**
6.1 Concept 45
6.2 Modified DCG Translation 45
6.3 Tracing Meta-Interpreter 48
6.4 Representing the Execution as Facts 52
6.5 Representing the Execution as a Dict 54
6.6 Provided Modules 57
6.7 Future Work 59

7 DCG Visualiser **61**
7.1 Implementation 61
7.1.1 Server 61
7.1.2 Client 62
7.2 Overview of the Application 63
7.2.1 Posing Queries 63
7.2.2 Interactive Visualisation 64
7.3 Future Work 66

8 Conclusion **67**
8.1 Summary 67
8.2 Outlook 67

List of Figures **69**

List of Listings **71**

Bibliography **73**

Appendix **79**
A.1 Screenshots of DCG Visualiser 79
A.2 CD Contents 87

1 Introduction

1.1 Motivation

The logic programming language PROLOG originated in 1972. It was a by-product of a project whose main subject was to process language [CR96]. Over time, PROLOG's applications have multiplied, but the ability to describe language and grammar with declarative rules is still one of its key features. For instance, PROLOG is used for natural language processing in IBM Watson [LF11]. There, using a parser, sentences are decomposed and further analysed.

Besides employing clauses to express grammars in PROLOG, there are also other options available. For example, it is possible to make use of regular expressions[1] or utilising the XML[2] schema definition language to validate XML data in SWI-PROLOG [NKS17]. One convenient way to describe lists in PROLOG are *Definite Clause Grammars (DCGs)*. These are an extension of context-free grammars and were introduced in 1980 by F. Pereira and D. Warren [PW80].

DCGS can be used in a variety of applications: besides the analysis and description of language, whether natural or formal, general operations on lists, which are essential in a multitude of programs, can also be described by them [Tri17].

The declarative syntax in logical programming is different from the imperative approach, whereas the latter is far more widespread. Also, the evaluation of queries in PROLOG, which makes use of backtracking and unification, differs from the execution with imperative programming paradigms. This is only one reason why learning logical programming in general can be intimidating and confusing [Tay88, PB87].

The notation of DCGS offers the possibility to express grammars with simple rules. This makes them a good choice for getting into declarative and logical programming. If knowledge exists, however, the functionality can be extended by nesting normal PROLOG code.

To illustrate complex processes and thus facilitate learning, tools for visualising are helpful. Existing tools for PROLOG can be utilised to some extent. However, the

[1] See library `pcre`, added to SWI-PROLOG version 7.6.0-rc1 in September 2017 [Wie17a].
[2] XML: Extensible Markup Language.

representation and available functions often do not suit DCGs well and generate unnecessary overhead. Therefore, there is a need for an appropriate application.

The goal of this work is to establish an interactive visualisation for DCGs. A visualisation can be useful both for beginners who learn how to work with DCGs, as well as for experienced users who want to debug their own code.

1.2 Structure of this Work

This thesis starts with an overview of related research in Chapter 2. With these insights into current possibilities to visualise Prolog and DCGs, the concept of a visualisation is to be introduced in Chapter 3. Based on a user scenario, relevant properties are derived. These are incorporated into the presented mock-up.

Subsequently, used technologies are introduced. These Prolog related topics are presented in Chapter 4. In addition to an introduction to DCGs, tracing in SWI-Prolog, storing structured data in *dicts*, and connecting SWI-Prolog to the web using the library *Pengines* is discussed.

Within the scope of this thesis, different ways to collect data on the execution of DCGs are considered. In addition to the usage of DCG's own capabilities, or other programming languages than Prolog to analyse the execution of DCGs, an approach using SWI-Prolog's built-in tracer to gather data, is discussed in Chapter 5. Concluding, approaches based on meta-interpreters are introduced there.

In Chapter 6, the implemented approach is presented in detail: This is a meta-interpreter written in Prolog, which is capable of gathering information on the execution of queries on a DCG. The following Chapter 7 gives an overview of the graphical user interface. This is realised as a web application and enables the interactive visualisation of DCGs.

The thesis concludes with a summary and an outlook on the continuation of this work in Chapter 8.

2 Related Work

In this chapter, an overview of existing work, which relates to visualising Prolog and DCGs, is given. In addition to tools for natural language analysis, there are already several approaches for the visualisation of Prolog. These are not specifically designed for DCGs, but provide an insight into how visualisation of logical programs can look and what the benefits of these systems are. Finally, selected examples of visualisations for other programming languages will be briefly presented.

2.1 Language Analysis in Prolog

A grammar kit in Prolog by K. Kahn in 1984 [Kah84] is an environment for natural language projects. With the help of a dialogue system, it offers children the opportunity to experiment with grammars in Prolog. Queries can be made to a previously specified grammar. If unknown words are read, the system will ask questions to the user such as `"Is (WORD 'dog' NOUN) true?"`. The answers will expand the dictionary, also an updated view of the current parse tree is provided. One key finding of this work is "That dynamic graphics is *(sic)* very valuable as a tool for observing and debugging complex processes" [Kah84, p. 185].

An extensive graphical environment for natural language processing is *HDRUG*, developed by G. Noord and G. Bouma in 1997 [VNB97]. In addition to the graphical output of different representations of structures and lexical entries, statistical information can be displayed as graphs. The extensibility of the system makes it possible to test different parsers against each other or to define new output formats. A sample view of a parse tree with additional information about lexical entries is given in Figure 2.1.

2.2 Visualising the Execution of Prolog

Since the early years of Prolog, there has been research on debuggers and visualisations. In the extensive survey on logic program analysis and debugging of M. Ducassé and J. Noyé from 1994 [DN94], an overview of existing approaches at

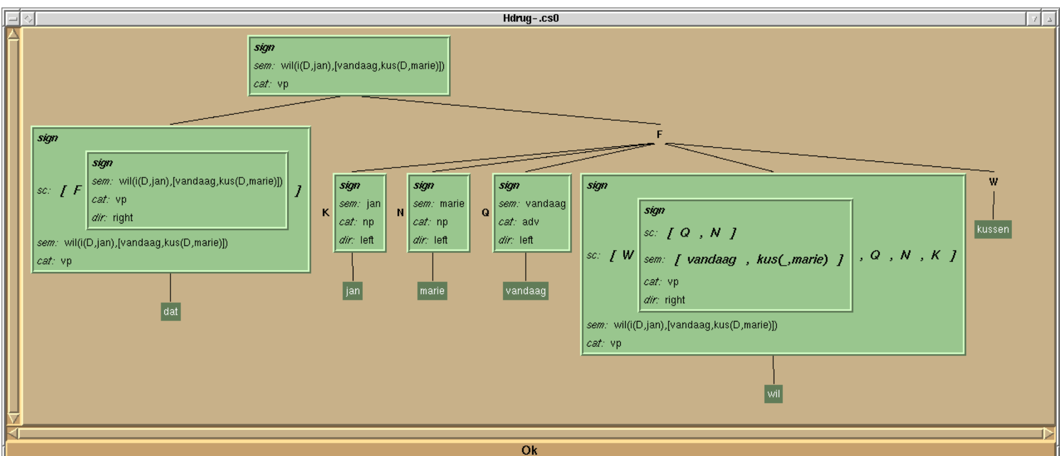


Figure 2.1: View of a parse tree with additional information in HDRUG [VN]

that time is given. This section of the thesis covers some quite old research as well as more recent approaches.

The *Transparent Prolog Machine*, introduced by M. Eisenstadt and M. Brayshaw in 1988 [EB88], is a PROLOG execution model, attached to a graphical debugger. Besides a fine granular tree of the execution, high-level trees with some abstraction can be displayed.

GRASP by H. Shinomi from 1989 [Shi89] parses programs to build graphs, representing the program structure and relations between rules. These are used by a step execution interpreter for visualising the program flow.

Another approach, which focuses on the SLD resolution[3], was proposed by D. Tamir et al. in 1995 [TAK95]. A step-by-step evaluation, in combination with user-defined breakpoints, is possible with the help of a visual debugger. In addition to that, the current SLD resolution tree is displayed. The user can interactively select a path in the tree for further traversing, manipulating the default program execution order.

S. Mallet and M. Ducassé created *MYRTLE* in 1999 [MD99] for deductive databases debugging. This is a meta-interpreter, which runs synchronised with a tracer for query execution analysis. The gathered data is displayed as an augmented SLD tree using GML[4], a predecessor of GraphML[5].

SLD trees for "real world" programs can become very complex and confusing. Therefore, they are only partially helpful for debugging. G. Kókai et al. introduced a *Graphical Interactive Diagnosing, Testing and Slicing System (GIDTS)* in

[3]SLD: Selective Linear Definite clause resolution [KK71].
[4]GML: Graph Modelling Language.
[5]GraphML: Graph Markup Language.

1999 [KNN99] to offer a suitable debugger. An interactive proof tree, which also contains failing branches, is used as a visualisation.

The *textual tree trace (TTT)* by C. Taylor et al. (1999) [TdBP99] focuses on text based tracing instead of a visualisation. It can be sorted between common textual tracers, like in SWI-Prolog, and graphical interfaces: The TTT notation is purely text based, but visualises a detailed view of the program flow, including call status information and current variable binding and unbinding.

To support students in learning Prolog, *PrettyProlog* was published by A. Stalla et al. in 2009 [SMM09]. This interpreter is written in Java and capable of showing the SLD tree, along with other tools for inspecting the program, like a view of the program stack.

Using puzzle pieces as a metaphor for the multiple subgoals which have to be satisfied for constructing a proof, L. Mondsheim et al. (2010) [MSL10] proposed an intuitive visualisation of logical deduction.

A further visualisation of SLD trees, although statically with LaTeX, is provided by *SLDNF-Draw* from M. Gavanelli (2017) [Gav17]. The tree is computed using a meta-interpreter. It is available as a SWI-Prolog package.[6]

The *Prolog Visualizer* by Z. Lai and A. Warth (2015) [LW15] is a web application which features an in-depth step-by-step execution of Prolog. In addition to visualising the chronological order of execution, current variable substitutions, failed subgoals and backtracking are displayed in an appealing manner. Unfortunately, it is not possible to use the tool with DCGs since it only supports a rather limited subset of Prolog.

Using *Constraint Handling Rules (CHR)*, a methodology to visualise the execution of Prolog is given by N. Sharaf et al. (2017) [SAF17]. It can be used to visualise intermediate computation steps. For example, in the case of a Sudoku solver, every change on the board can be displayed, visualising the search strategy and effects of backtracking when failure occurs.

In addition to the presented research, some Prolog implementations also involve debugging tools. SWI-Prolog, which is used in this work, features multiple options for inspecting programs: Besides the textual tracer a graphical tracer, an execution profiler for retrieving statistical data, a cross referencer for analysing dependencies, and a navigator for browsing the project structure are available [WSTL12].

[6] SWI-Prolog package *sldnfdraw:* `http://www.swi-prolog.org/pack/list?p=sldnfdraw`.

2.3 Interactive Visualisations for Other Programming Languages

A comprehensive overview of interactive visualisations for other languages would substantially exceed the scope of this work, hence only two examples will be given.

ClickyEvaluation, developed by S. Kögel et al. (2016) [KCT16], is an evaluator for functional programming in Haskell. Utilising a web application, the user can determine the order of the evaluation and thus evaluate the query step by step. It also provides a history of previous steps, with the possibility for returning to any of these snapshots.

Ohm, “a library and language for building parsers, interpreters, compilers, etc” [Wpc], is based on *Parsing Expression Grammars*. The grammar notation is fairly comparable to DCGs. Besides a tracer on command line level, there is also a graphical visualiser [Ohma, Ohmb] which offers a good overview of which sub strings of the given input matches against according rules.

3 User Scenario

As it can be deduced from the presented overview of related work in Chapter 2, there is currently no visualisation that suits DCGs well. To change this, an application for the visualisation of DCGs is described in this chapter. Based on user requirements, which are discussed in Section 3.1, application components are introduced in Section 3.2. The chapter concludes with a summary of the intended application.

3.1 Target User Groups

To determine what features are needed, it is important to assess which requirements users will have. In the scope of this work, we imagine two potential user groups. On the one hand, there are *beginners*, who just start to discover DCGs for the first time. On the other hand, we assume more *experienced users*, who want to debug and fully understand their own programs.

3.1.1 Beginners

Beginners are users who want to get started with DCGs and want to understand how these are interpreted. Due to the limited knowledge, it can not be taken for granted, that these users are capable of writing grammars by themselves. Using existing grammars, they should be aided in comprehending how a DCG is executed and thereby understand the returned results.

These users are to be supported to understand how PROLOG produces the result. It should be shown how the proof of a solution is gradually built up. Furthermore, it should be possible to display the execution steps PROLOG performed, which later became void by backtracking.

The visualisation should be as intuitive as possible. For a grammar rule it should be easy to see which sub-calls were triggered. It should also be clear which parts of the input are represented by each rule. This applies both to the intermediate steps and to the presentation of the result.

3.1.2 Experienced users

We call those experienced users who are able to create DCGs themselves. This could be advanced novices, as well as long-term Prolog users. One reason to utilise a visualisation is for debugging when problems are encountered.

A typical problem with Prolog is that failing queries just return `false` without further explanation. If a grammar should accept an input, but this is not the case, this answer is not helpful. Another scenario are queries to a DCG which do not terminate, due to left-recursion. By visualising the execution steps, the users can be given the possibility to see how Prolog executed a query and why it behaved faulty.

Depending on the complexity of the grammar, a comprehensive presentation of all execution steps at once can become obscure. Therefore, there is a need for a simple and clear presentation. In addition, it is desirable to let the user decide whether additional information should be displayed or not.

Advanced grammar components such as embedded Prolog code or semicontext extend the application possibilities of DCGs. The visualisation should provide the opportunity to use and visualise these components, or ensure the extensibility to add corresponding features later.

3.2 Application Mock-up

Based on user requirements and inspired by existing research, a concept of an application is presented below. First, the input options for DCGs and queries are discussed. Subsequently, the basic representation of the execution steps and the interactive features such as additional presentation options and step-by-step execution are introduced.

3.2.1 Enter Grammars and Run Queries

To execute a query, the user has to specify a corresponding grammar first. If desired, a sample grammar can be selected from a list of examples. This is then loaded into the text field and can be modified as desired. Alternatively, grammar rules can be entered from scratch.

After that, queries can be formulated. For this, three given input fields have to be filled. Besides the grammar body, an input and remainder list have to be stated. Since a DCG can also be utilised for the generation of outputs, it should be possible

to specify variables as parameters. This can be achieved by using PROLOG syntax, e.g. by beginning the variable identifier with an upper-case letter.

Initiating the execution is handled by two buttons: *Run Query* for getting the first result, and *Next Solution* to retrieve further solutions, if these exist. See Figure 3.1 for a graphical representation of the features described above.

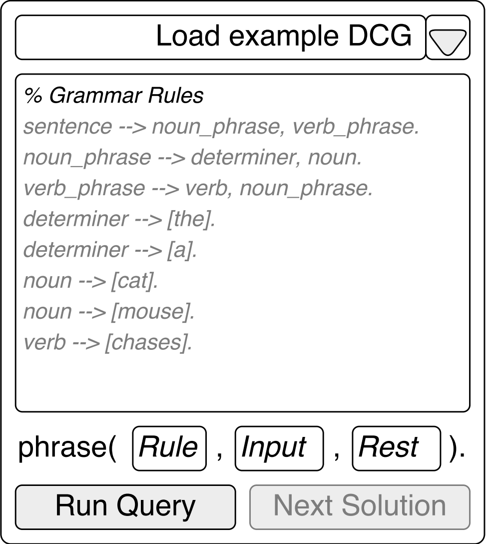


Figure 3.1: Entering a DCG and a query

3.2.2 General Idea of Visualising the Result

In contrast to some of the presented visualisations in Chapter 2, the focus is not on the representation of SLD trees. Instead, the visualisation is based on parse trees. In its simplest form, grammar rules can be read as *'A rule is satisfiable, provided all elements in the rule's body can be satisfied'*. This leads directly to the representation by means of a parse tree, in which subgoals lead to the satisfaction of a goal. This has the advantage that the result can be represented in a clear manner, as in Figure 3.2.

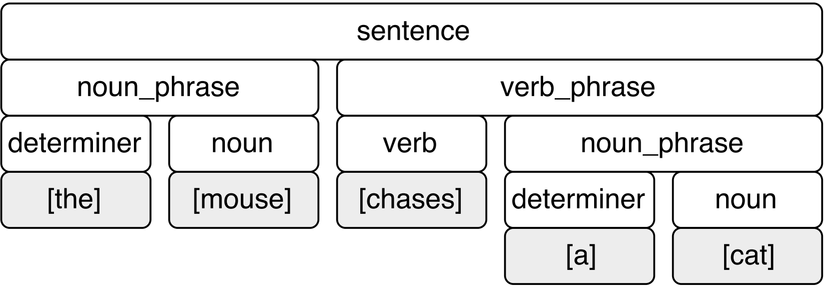


Figure 3.2: Displaying a result

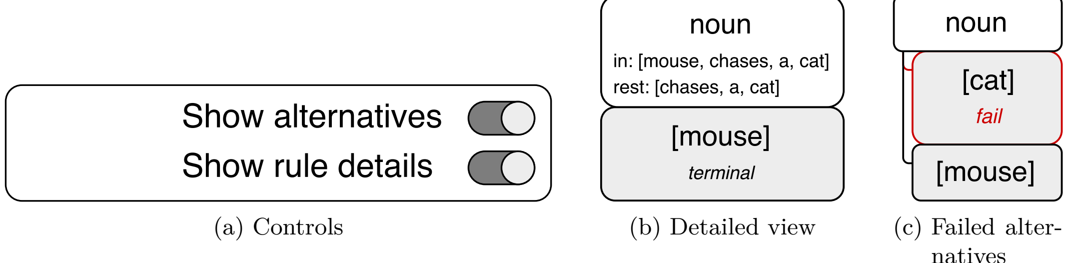
Instead of using a common tree structure, boxes are used for the visualisation of the underlying tree. The structure can be read from top to bottom, just like parse trees: E.g. direct child nodes are displayed side by side below a goal and terminals are stored in the leaves. The horizontal extent of the boxes also makes it easy to see which terminals are assigned to a goal, even if the assignment takes place in child nodes.

In the case of left-recursion, which result in non-terminating executions, an output should nevertheless be possible. This can be achieved by depth-limiting the execution, coupled with hints in the visualisation that the given execution reached the allowed recursion limit.

3.2.3 Integration of Additional Information

The representation of the query's result as a parse tree hides failing goals. However, these will be needed to explain failure. For this reason, execution branches, which have been neglected during backtracking or failed without further alternatives, should be displayed. The resulting structure is similar to an *AND/OR-Tree* [PB87], which also displays alternative computation branches. Here, instead of next to each other, executed alternatives are displayed one above the other. If unwanted, it should be possible to hide these discarded execution steps, displaying only the result of a query as a parse tree.

Another possibility to adjust the visualisation is by showing additional information about a goal: Details, such as unifications at runtime of the goal and its input and rest lists or additional information about embedded PROLOG code inside a rule, can be displayed.



(a) Controls (b) Detailed view (c) Failed alternatives

Figure 3.3: Toggle additional information

In Figure 3.3 is an example of a detailed view and the visualisation of a failed goal for which a solution was later found through backtracking. The controls give users the choice, how much details should be presented. This allows for multiple views of the result: A quick overview of the results as well as detailed insights into execution

steps can be shown. In the current concept of the application, these settings are global. However, it is conceivable to extend this and enable showing details and failing branches only for individual goals.

It is possible that only a small part of the execution is of interest, for example if an error has been localised and is to be tracked. Through provided buttons, it is possible to zoom in and view a part of the visualisation in detail.

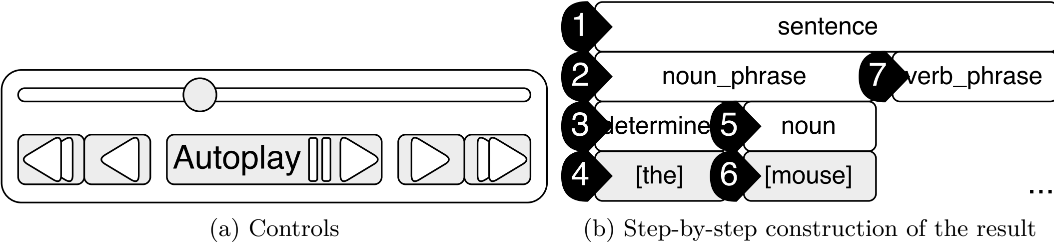
3.2.4 Visualising Execution Order

It is important to show the order in which a result is satisfied. A step-by-step visualisation can help understand how PROLOG executes a query. Gradually, invoked goals are added. Unsatisfiable goals result in backtracking, visualised by creating alternative branches in the parse tree. For the previous example of a visualised parse tree (see Figure 3.2), the sequence of fade-in is indicated in Figure 3.4.

In this case, it is desirable to offer the possibility to jump back to earlier points of the execution. This ensures that difficult parts can be viewed multiple times directly without having to restart each time from scratch. The desired controls are a slider, which allows a fast selection of any point in the execution history, buttons for viewing the execution steps one-by-one, the possibility to get to the last or upcoming solution with a single click and possibly a continuous play feature, to start or pause an automatic playback of the execution.

Using colours, the current status of each goal can be indicated. This allows for an easy distinction whether a goal is currently running, void through backtracking, succeeded, or failed.

Besides the visualisation of execution steps, it would be handy to see directly, which rule is currently used for reasoning. This can be achieved by highlighting the corresponding code section for each execution step.



(a) Controls (b) Step-by-step construction of the result

Figure 3.4: Showing an execution step-by-step

3.3 Summary

An overall view of the intended application, based on the previously introduced features, is presented in Figure 3.5. The user interface is divided into two main areas: On the left side, there are user inputs and controls whereas the visualisation is on the right. The currently evaluated query, including resulting unifications, is briefly summarised above the actual visualisation. To adjust the zoom level of the parse tree, there are corresponding buttons.

The application is tailored for an easy to read representation of the execution result, bundled with interactive features to show its progress. Also, the degree of displayed details can be controlled by users to fit their needs. Hereby a tool that can be helpful when handling DCGs, both for beginners and experienced user, is provided.

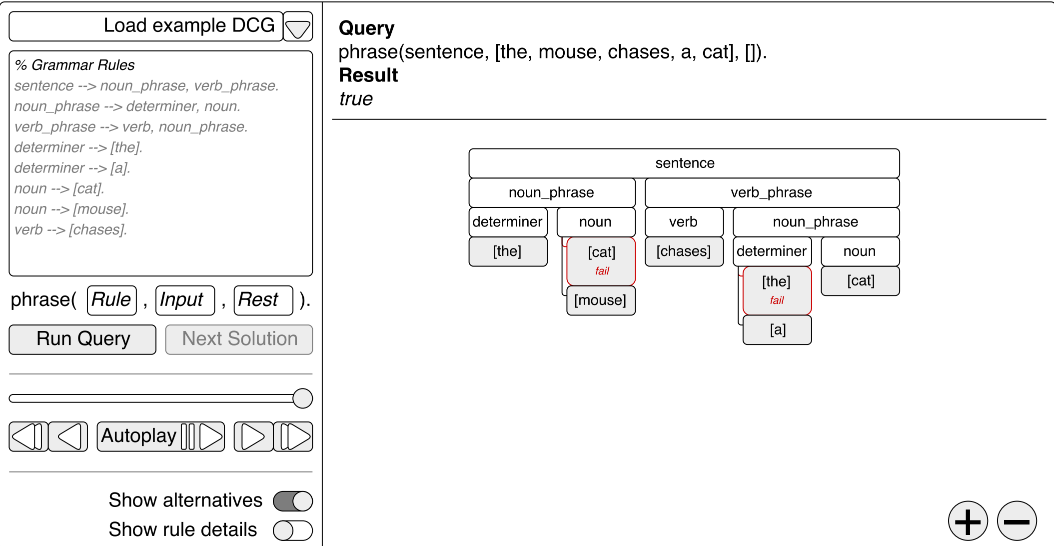


Figure 3.5: Overview of the intended application

4 Technology: Prolog

After a brief look at the history of Prolog and the used implementation, SWI-Prolog, this chapter covers several Prolog related technologies that are relevant to this work. First, DCGs are presented. Subsequently, the execution model and tracing in SWI-Prolog is explained, followed by a presentation of the data structure *dict* and the library *Pengines*.

4.1 Origins of Prolog

In 1970, Alain Colmerauer (⋆1941, †2017) began working on a project with the aim to process natural language using a computer. Together with Robert Pasero and Philippe Roussel, first steps were taken by creating non-deterministic context-free analysers and a French paraphrase generator. These were implemented in Algol 60 and Q-systems. The latter is a programming language Colmerauer created in earlier research, aimed at creating a high-level language. It was equipped with rewriting rules and a limited version of unifications [CR96].

The team was extended by Jean Trudel, who worked on automated theorem proving. Thus in 1971, a first natural-language communication system emerged. Simple conversations with the computer in French were possible. After entering some sentences, questions could be asked. Reasoning based on the given data was possible with a developed prover, as well as an output in French.

One of the researchers who invented the SLD resolution [KK71], Robert Kowalski, supported the team. After some research, linear resolution with unification only between the head of clauses was considered as a resolution strategy. Using this, the first Prolog system was implemented in 1972 in Algol-W. At the same time, the team developed a man-machine communication system, using Prolog. Also, the language was given its name, Prolog, which is short for *Programmation en Logique*. In the following years, syntax, primitives and the computing method were further refined. These essential features still form the basis of the current version of Prolog.

In the following years multiple language dialects arose, later Prolog was standardised in 1995 as ISO/IEC 13211-1 [ISO95]. Currently, several implementations are available. Within this work, SWI-Prolog [WSTL12] is used. This implementation

was first published in 1987 and is mainly developed by Jan Wielemaker. The current stable release at the time of writing was utilised, this is version 7.6.0, released in October 2017. SWI-Prolog provides a rich set of built-in predicates and libraries. These make it easy to connect Prolog to the web. Since the visualisation is implemented as a web application, this is a good fit.

4.2 Definite Clause Grammars

The idea of *Definite Clause Grammars* was published in 1980 by Fernando Pereira and David Warren. They evolved from Colmerauer and Kowalski's *metamorphosis grammars* (1975) [PW80]. DCGs are considered to be included in the ISO Prolog standard and currently available as a draft [Dod92][Hod15].

DCGs are grammar rules for describing difference lists. They can be used for tasks like language processing, both natural or formal. This can be achieved, due to the fact, that any text is a list of chars. Compared to context-free grammars, they have multiple extensions:

- **Providing context-dependency** through the use of additional arguments on non-terminals.
- **Building custom parsing structures** while querying a DCG. This is discussed later in Section 5.1.
- **Allowing additional conditions** in the grammar rules through embedding arbitrary Prolog code.

A quick look at the capabilities of these grammars is made in the introductory example in Section 4.2.1. After that, the syntax of DCGs is introduced in Section 4.2.2. DCGs are not directly evaluated, SWI-Prolog does some preprocessing on grammar rules, resulting in normal Prolog clauses. These expansion rules are explained in Section 4.2.3, followed by an overview of the built-in predicates for querying DCGs in Section 4.2.4.

4.2.1 Introductory Example

In order to get an initial insight into DCGs, basic grammar components are presented in the following using an example: A simple grammar for English sentences is to be defined. The representation of a sentence is a list of atoms, the individual words. For the sake of simplicity, spaces and punctuation marks are neglected.

Sentences like

`[a, cat, chases, the, mouse]` or
`[the, mouse, chases, a, mouse]`

should be able to be represented with our grammar. In our example, a `sentence` can be formed by a `noun_phrase`, followed by a `verb_phrase`. This can be formulated as a DCG rule as follows:

```
sentence --> noun_phrase, verb_phrase.
```

Similarly, a `noun_phrase` is formed by a `determiner`, followed by a `noun`. Possible phrases which are represented by a `determiner` are `the` and `a`.

```
determiner --> [the].
determiner --> [a].
```

Together, these rules form a DCG for some basic sentences. The entire example DCG is given in Listing 4.1.

Listing 4.1: DCG for a subset of the English language

```
1  % structure
2  sentence    --> noun_phrase, verb_phrase.
3  noun_phrase --> determiner, noun.
4  verb_phrase --> verb, noun_phrase.
5
6  % vocabulary
7  determiner --> [the].
8  determiner --> [a].
9  noun --> [cat].
10 noun --> [mouse].
11 verb --> [chases].
```

The basic form of a grammar rule is

```
Head --> Body.
```

This means, the rule named *Head* can be rewritten by its grammar body *Body*. A call of a grammar rule succeeds, if a provided list unifies with the concatenated lists from its body items.

Valid with respect to `sentence` are lists like

`[a, cat, chases, the, mouse]`,
`[the, mouse, chases, a, cat]`, or
`[the, mouse, chases, the, mouse]`.

An example for a list which is not accepted by the grammar is given below:

`[the, dog, likes, the, cat]`

This is invalid, since `dog` is no valid `noun` and only `chases` is allowed as a `verb`. In order to accept this list as well, one would have to extend the defined vocabulary accordingly.

As usual in PROLOG, grammar rules can be queried with unbound variables, or lists of arbitrary values. This allows the usage of a DCG as a generator of valid outputs, as well as a way to check, whether a given query is allowed in the scope of a grammar rule.

The DCG from Listing 4.1 is very limited in its capabilities of representing English sentences, e.g. it lacks the support of plural. One way of adding this feature is to add an attribute to each of the rules which describes the grammatical number and extend the vocabulary by plural forms of the words [Fla94, p. 139]. The adjusted DCG is given in Listing 4.2.

Listing 4.2: Modified English grammar with grammatical number

```
1  % structure with grammatical numbers
2  sentence(N)    --> noun_phrase(N), verb_phrase(N).
3  noun_phrase(N) --> determiner(N), noun(N).
4  verb_phrase(N) --> verb(N), noun_phrase(_).
5
6  % vocabulary
7  determiner(_)        --> [the].
8  determiner(singular) --> [a].
9  determiner(plural)   --> [].
10
11 noun(singular) --> [cat].
12 noun(plural)   --> [cats].
13 noun(singular) --> [mouse].
14 noun(plural)   --> [mice].
15
16 verb(singular) --> [chases].
17 verb(plural)   --> [chase].
```

Now, there are singular and plural forms of each word in the vocabulary. These can be distinguished by the attribute which was added to each non-terminal. Accordingly, rules which describe the structure of sentences are altered. For instance, a sentence is still a `noun_phrase`, followed by a `verb_phrase`, but now with the same grammatical number. This is ensured by the shared variable.

This extends the scope of the grammar, so `sentence` accepts also lists like

`[cats, chase, mice]` or
`[the, mice, chase, the, cat]`.

Utilising the basic grammar rule components presented here, it is already possible to formulate comprehensive grammars. In addition, there are other features which can be used to describe grammars. This is presented in the next section.

4.2.2 Syntax of DCGs

DCGs are not limited to the syntax and functionalities shown in Section 4.2.1. A full overview is given in this section.

One way to specify the syntax of DCGs is to use the grammar rules itself. A corresponding grammar is given in Listing 4.3. This informal definition of DCGs is based on [CM03, p. 230], but extended to match the current ISO Prolog draft of *definitive clause grammar rules* [Hod15]. To ensure easy readability of the rules, here, white-space characters between these building blocks are skipped in the rule definition. These have no syntactic meaning and can be used as desired, just like in regular Prolog code.

Listing 4.3: Syntax of DCGs

```
1 definite_clause_grammar --> grammar_rule.
2 definite_clause_grammar --> grammar_rule, definite_clause_grammar.
3
4 grammar_rule --> grammar_head, ['-->'], grammar_body, ['.'].
5
6 grammar_head --> nonterminal.
7 grammar_head --> nonterminal, [','], semicontext.
8 semicontext  --> terminal.
9
10 grammar_body --> grammar_body, [','], grammar_body.
11 grammar_body --> grammar_body, [';'], grammar_body.
12 grammar_body --> grammar_body, ['->'], grammar_body.
13 grammar_body --> grammar_body, ['|'], grammar_body.
14 grammar_body --> ['\+'], grammar_body.
15 grammar_body --> grammar_body_item.
16 grammar_body --> ['('], grammar_body, [')'].
17
18 grammar_body_item --> nonterminal.
19 grammar_body_item --> terminal.
```

```
20 grammar_body_item --> ['!'].
21 grammar_body_item --> ['{'], prolog_code, ['}'].
22
23 % terminal:       List
24 % nonterminal:    A//N
25 % prolog_code:    Arbitrary prolog code
```

A *definite clause grammar* is a set of multiple *grammar rules*, each with the form:

`Head --> Body.`

The rule's head is either a *non-terminal* or a non-terminal followed by a *semicontext*. The semicontext, sometimes also referred to as *pushback list*, is a list, which is prefixed to the remaining terminal list after successfully evaluating the grammar's body [Hod15].

Non-terminals are callable terms which reference to a grammar rule. For indication, the *non-terminal indicator* `A//N` can be used. `A` is an atom, the name of the rule, and `N` is a non-negative integer which states its arity.

Terminals are Prolog lists. The individual list items can be of any type, including variables. Internally, strings are handled as lists of chars, so strings can also be used instead of explicit lists. There is no short form as `-->/1` for an empty body. However, a rule which describes an empty string can be built by an empty list:

`Head --> [].`

Inside the rules body, there can be various *grammar body items*, such as terminals, non-terminals, the cut `!/0`, or normal Prolog code surrounded by curly brackets `{..}`.

Also, the Prolog control predicates for *conjunction* `,/2`, *disjunction* `;/2`, *If-Then/-Else* `->/2`, *disjunction (deprecated form)* `|/2`, and *not* `\+/1` can be used to express relationships between grammar body items. These control predicates behave as in regular Prolog. *Parentheses* `( .. )` can be utilised as usual.

4.2.3 DCG Expansion in SWI-Prolog

The DCG specification defines only syntax and semantics, but no concrete implementation. Although there is a reference implementation available [MD10], using it is not mandatory. Diverse approaches are conceivable: For instance the interpretation of the `-->/2` operator at runtime, or a translation of the DCG rules in normal Prolog code, prior to execution. The latter is used by SWI-Prolog [WSTL12] and is therefore presented here.

When loading code into SWI-Prolog, the compiler calls `expand_term/2` on each term read from the input. This leads to four preprocessing steps:

1. Handling of conditional compilation directives.
   See documentation [Wie17b, pp 95–96] for more information.
2. `term_expansion/2`: Execution of user-defined term-expansions.
3. `dcg_translate_rule/2`: Translation of a grammar rule into a normal Prolog clause.
4. `expand_goal/2`: Call expansion on terms which appeared in the output of the previous steps. This applies expansions until there are no more changes.

During the expansion of grammar rules, non-terminals are extended with two additional arguments. These build a difference list and are appended to possibly already existing arguments. The first added argument represents the input list, the second is the remaining list after a rule has been executed. Similarly, a terminal `[T]` is expanded to `Input = [T|Rest]`. Successive, grammar body items are linked together, e.g. the remainder of one item becomes the input of the following.

By viewing a listing, these expanded rules can be inspected. Grammar rules from Listing 4.1 are converted by SWI-Prolog as follows:

```
?- listing(sentence//0).
sentence(A, C) :-
  noun_phrase(A, B),
  verb_phrase(B, C).

?- listing(determiner//0).
determiner([the|A], A).
determiner([a|A], A).
```

Note, that SWI-Prolog does slightly optimise the DCG: For instance, instead of expanding the first rule of `determiner//0` to

```
determinder(A, B) :-
  A = [the|B].
```

it is asserted as a fact, nesting the functionality inside the predicates head. This allows for faster execution due to SWI-Prolog's rule indexing capabilities. Clauses whose arguments can not be unified with a query are not considered as candidates in the execution [Wie17b, p. 58].

Prolog code, inside curly brackets, is copied as is, followed by an unification of the predecessors remaining list with the successors input list:

```
?- dcg_translate_rule((rule --> {any}), Translated).
Translated =  (rule(A, B) :- any, A=B).
```

Also, according to the semantics, a given semicontext is prefixed to the remainder before terminating the goal:

```
?- dcg_translate_rule((rule, [semicont] --> [rulebody]), Translated).
Translated =  (rule([rulebody|A], [semicont|B]) :- true).
```

Control structures, which can be used inside a grammar body, are adopted in the same way as stated in the grammar rule.

The aim of this work is to visualise DCGs. Within SWI-Prolog, however, by default, only the expanded form is available. Therefore, some approaches to analyse the execution of DCGs in Chapter 5 are based on the interpretation of this translated form.

4.2.4 Querying DCGs

There are two built-in predicates in the ISO Prolog draft which can be used for calling DCGs: `phrase/2` and `phrase/3` [Hod15]. The latter accepts three arguments:

```
phrase(:DCGBody, ?Input, ?Rest).
```

The `DCGBody` is usually only a non-terminal, but is not restricted to this and could make use of the language features for `grammar_body` as stated in Section 4.2.2. `Input` is the list, which is handed to the grammar body, while `Rest` is the remainder after calling the grammar body with given input.

As an example, querying the DCG from Listing 4.1, the `noun_phrase//0` corresponds to `[the, cat]` from the input list, the remaining list is unified with the third argument of `phrase/3`.

```
?- Input = [the, cat, chases, a, mouse],
   phrase(noun_phrase, Input, Rest).
Input = [the, cat, chases, a, mouse],
Rest  = [chases, a, mouse] .
```

The short form `phrase/2` expects an empty remaining list and therefore can easily be expressed as a call of `phrase/3`:

Listing 4.4: Declaration of `phrase/2`

```
1 phrase(DCGBody, S) :-
2   phrase(DCGBody, S, []).
```

Of course, calling a DCG is not limited to already instantiated lists. Both input and rest list can contain or be uninstantiated variables. This makes it possible to use a DCG for both verification and generation of lists with these predicates.

4.3 Tracing in SWI-Prolog

Since the standard does not specify, there can be differences in the execution model depending on the PROLOG implementation. The common 4-port Byrd Box model, introduced by L. Byrd [Byr80], is extended by several additional ports in the SWI-PROLOG execution model [WSTL12, p. 27]. In the following, first the model is discussed, succeeded by an overview of the command line tracer and a way of intercepting the tracer.

4.3.1 Execution Model

In the 6-port box model, the control flow of a PROLOG program can be imagined using boxes which are linked together. Each box represents a goal. The events during the execution are handled by the given ports. These are also the connections between goals and visualise the control flow. The graphical representation of a single goal is displayed in Figure 4.1.

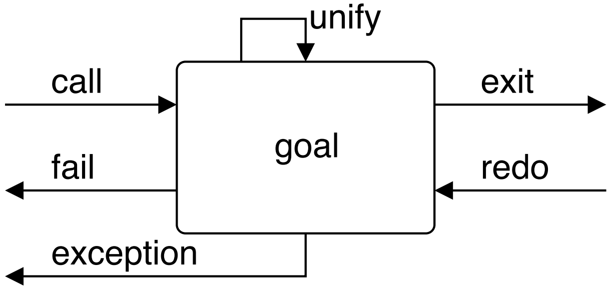


Figure 4.1: The 6-port execution model

Through the two additional ports in SWI-PROLOG, *unify* and *exception*, there is a total of six ports and their corresponding events. The meaning of these events is described in the following list.

- **call**
  A goal is called and PROLOG starts trying to satisfy it.
- **unify**
  Link between head-matching progress and invoking the rule's body. After calling a goal or invoking a redo, PROLOG unifies the goal with the next appropriate clause's head in the database. Then its body is executed.
- **exit**
  Goal has been satisfied. Linked to the port *call* of the next pending goal. If there is no pending goal, the execution is successfully finished.
- **redo**
  Try finding an alternative solution to goal. Invoked if a goal failed or a further solution is searched and there are left choice points in this goal.
- **fail**
  Goal could not be satisfied. If possible a *redo* is initiated on this, or a preceding goal with left choice points.
- **exception**
  An exception is raised. This port is activated on each preceding goal until the exception is caught.

Subgoals, which are called within a clause's body, can be imagined as nested inside its corresponding parent box. These have to be satisfied in order to satisfy their parent goal.

4.3.2 Command Line Tracing

SWI-PROLOG features a command line tracer for debugging purposes. It can be enabled by calling the built-in predicate `trace/0`. Until disabling the debugging mode, e.g. by calling `nodebug/0`, SWI-PROLOG will trace goals: Occurring events are written to the command line.

By default, all visible ports are *leashed ports*. This means, after printing the current port, the user is prompted for an action. This gives the user the opportunity, amongst others, to step (*creep*) into the next goal, or skip the tracing of subgoals. Leashing can be adjusted using the built-in `leash/1`.

As an example, the output of the execution of a query on a previously introduced DCG (see Listing 4.1) is given below. Each line represents one traced execution step naming the current port, the recursion level, and the current goal. This is followed by the user prompt, here, each answered with *creep* to show the full execution.

```
[trace]  ?- phrase(noun_phrase, NP).
  Call: (11) noun_phrase(_6874, []) ? creep
  Call: (12) determiner(_6874, _7148) ? creep
  Exit: (12) determiner([the|_7134], _7134) ? creep
  Call: (12) noun(_7134, []) ? creep
  Exit: (12) noun([cat], []) ? creep
  Exit: (11) noun_phrase([the, cat], []) ? creep
NP = [the, cat] ;
  Redo: (12) noun(_7134, []) ? creep
  Exit: (12) noun([mouse], []) ? creep
  Exit: (11) noun_phrase([the, mouse], []) ? creep
NP = [the, mouse] ;
% ...
```

By default, all ports introduced in Section 4.3.1, except of the port *unify*, are visible by the tracer. Using `visible/1`, the visibility of them can be adjusted. This has been done in the subsequent example. Now, the difference between the first and second call of `noun//0` can be distinguished throughout the execution, not just by comparing the results. Using a listing inside the tracer (invoked by the command *L*) the current clause is listed. This is followed by another prompt of the current goal waiting for further user commands.

```
[trace]  ?- visible(+unify), leash(+unify).
true.
[trace]  ?- phrase(noun, N).
  Call: (11) noun(_2780, []) ? creep
  Unify: (11) noun([cat], []) ? Listing noun([cat], []).
  Unify: (11) noun([cat], []) ? creep
  Exit: (11) noun([cat], []) ? creep
N = [cat] ;
  Redo: (11) noun(_2780, []) ? creep
  Unify: (11) noun([mouse], []) ? Listing noun([mouse], []).
  Unify: (11) noun([mouse], []) ? creep
  Exit: (11) noun([mouse], []) ? creep
N = [mouse].
```

The tracer provides a detailed insight into the execution of a query. However, the textual presentation quickly becomes confusing in the case of extensive queries. In addition to the command line tracer, SWI-Prolog also offers a graphical tracer. Although this is also not specifically optimised for use with DCGs, it offers additional possibilities, such as a view of the call stack.

4.3.3 Intercepting the Tracer

The output of the tracer shown in Section 4.3.2 is human-readable, but not especially suited for further program based analysis. However, SWI-Prolog offers the possibility to redefine the tracer itself. The predicate `prolog_trace_interception/4` is a built-in hook which serves this purpose. If this predicate has been defined by the consulted program, it is called from the SWI-Prolog debugger before writing the trace statements to the command line. Provided it succeeds, the built-in text-based tracer is intercepted [Wie17b, pp. 512–516].

The predicate `prolog_trace_interception(+Port, +Frame, +Choice, -Action)` has the following arguments:

- **Port** Is the current port with which the tracer was called, as introduced in Section 4.3.1. There are also some additional ports for handling breakpoints and the invocation of a cut: `break`, `cut_call` and `cut_exit`.
- **Frame** Represents a reference to the current stack frame. The associated context can be analysed using the predicate `prolog_frame_attribute/3`.
- **Choice** Is a reference to the current choice point. There is also a corresponding predicate for further analyses of the context of the current choice, `prolog_choice_attribute/3`.
- **Action** Specifies how the execution should proceed. Similar to the command line tracer there are several possibilities, like `continue` which is equivalent to showing each execution step on the command line, or `skip` which executes the current goal without tracing.

Both the reference to the current frame and choice point can be used to get additional context information. A variety of information can be output utilising the predicate `prolog_frame_attribute(+Frame, +Key, :Value)`. This overview of possible values for `Key` and their associated `Value` is limited to those used later in Section 5.3. A comprehensive list can be found inside the documentation [Wie17b, pp. 512–513].

- **goal** The current goal. If the module of the current predicate differs from the calling context, it is represented as `module:goal`.
- **clause** A reference to the currently executed clause. There are several built-in predicates which can make use of this reference, like `clause/3`, for retrieving the clauses head and body, or `clause_property/2` which provides properties like the line number of a clause.

- **level** The recursion level of the frame.
- **parent** A reference to the parent local stack frame.

Using the predicate `prolog_choice_attribute(+ChoicePoint, +Key, -Value)`, it is possible to retrieve information about the current choice point. Again, there are several possible values for `Key`:

- **parent** A reference to the first older choice point.
- **frame** A reference to the corresponding frame of the provided choice point.
- **type** Further information about the type of the choice point. There are several possible types like `clause` if there are alternative clauses, or `jump` for in-clause choice point.

Using the predicates presented, it is possible to build a tracer which can be adapted to fit application-specific requirements. This is discussed in Section 5.3.

4.4 Dicts

There are several ways to represent structured data in PROLOG. Besides the possibility of describing a structure using arbitrary terms there are also more advanced options available. One advantage of utilising already existing approaches is that additional functions for working with these data structures are usually provided. For example, there is a designated representation for RDF triples [WSTL12] or the *field notation* (Seipel et al. [Sei02]) which provides declarative querying mechanisms.

An approach for describing abstract objects with a modern syntax for various uses are *dicts*. These were added to SWI-PROLOG in Version 7 (2014) [Wie14]. A dict is a term with the following syntax:

```
Tag{KeyA:ValueA, KeyB:ValueB, ...}
```

The `Tag` is either a variable or an atom, it can be used to name the type of the dict. Inside the curly brackets can be an arbitrary amount of key-value pairs. Each `Key` is either an atom or an integer and has to be unique within the dict. The associated `value` can be any valid SWI-PROLOG term. Thus, a value could also be a variable or another dict. This allows for nesting dicts inside each other. The order of these key-value pairs is without significance, internally, the dict is transformed into a representation which does not respect order. Dicts can be unified as normal terms. However, the order of the key-value pairs inside these dicts also makes no difference.

Using the infix operator *dot* `./2`, it is possible to obtain values from a dict:

```
?- Value = point{x:3, y:5}.Key.
Value = 3,
Key = x ;
Value = 5,
Key = y.
```

The dot operator can also be used to call *functions*[7] on a dict. Predefined are functions for getting and setting values. It is possible to declare additional functions. These are normal PROLOG predicates and can be defined by using the `:=/2` operator. An example is given in Listing 4.5.

Listing 4.5: User-defined functions on dicts

```
1 P.move(X, Y) := point{x:X0, y:Y0} :-
2   X0 is P.x + X,
3   Y0 is P.y + Y.
4
5 P.distance() := D :-
6   D is sqrt(P.x**2 + P.y**2).
```

Now, these functions can be executed on a given dict:

```
?- N = point{x:2, y:3}.move(4,1).
N = point{x:6, y:4}.

?- D = point{x:3, y:4}.distance().
D = 5.0.
```

As shown, functions can be defined in a way that they return dicts themselves as well as functions that return arbitrary terms. With this universal extensibility of functionality and the appealing syntax, they are suitable for numerous applications. In the case that an application needs to exchange data between PROLOG and JavaScript, the library *http/json* contains predicates for translating dicts into JSON[8]. This allows an almost seamless integration of dicts in web applications.

[7] SWI-PROLOG 7 introduced some new features that are not ISO PROLOG compatible, which lead to controversial discussions [Wie, Neu15]. Dicts are only one of those extensions. These reassign the meaning of the `./2` operator. Also unusual in the context of PROLOG, predicates that use the `./2` operator on dicts are actually called *functions* [Wie17b, p. 246].

[8] JSON: JavaScript Object Notation.

4.5 Pengines

Pengines (T. Lager, J. Wielemaker 2014 [LW14]) offers the possibility to use logical programming in web-based projects. It is used to connect SWI-Prolog to the visualisation, which was realised as a web application. It is based on SWI-Prologs libraries for threads, HTTP clients and server to implement an universally applicable high-level interface between SWI-Prolog and clients.

For the exchange of data between Prolog and JavaScript there are also alternatives. In the case of SWI-Prolog, several approaches are conceivable. It features libraries for handling web protocols [WHVDM08] which could be used to create a server with a JSON API[9]. Alternatively, there are packages available [KSS+, Laa] which make using SWI-Prolog possible within a Node.js environment.

Here, we choose Pengines, as it has been designed to fit the requirements of multi-user scenarios as a web application. An overview of structure and functionality of Pengines is given in Section 4.5.1. This is followed by a discussion of built-in security features in Pengines.

4.5.1 Usage of Pengines

A *Pengine (Prolog Engine)* is a thread on a Pengines Server. On the server, libraries and predicates can be provided which are accessible for each Pengine. When a Pengine is created, Prolog code can be passed, which then can be used within it for querying. Each Pengine lives inside its own thread and module. Thus, several Pengines can exist concurrently with individual available clauses on the server.

The creation and access of a Pengine is possible using a client or SWI-Prolog itself. Currently, an implementation of a client library is only available for JavaScript and SWI-Prolog, the implementation for other languages is pending.

The communication between the client and server takes place by means of queries formulated in Prolog. When using the JavaScript client, results are transferred by default in JSON format. Although a Pengine can also be used for multiple queries, the default life cycle is stateless. Immediately after creation, a request is made, one or more results are fetched, and then this Pengine is destroyed.

A minimal working example of the client side of a Pengine is given in Listing 4.6. To use Pengines, `pengines.js` provides an API. This requires jQuery, so this is also loaded. Subsequently, a Prolog program is given in a script tag with the type `"text/x-prolog"`. In principle, any code can be stated which then can be used

[9] API: Application Programming Interface.

within the Pengine. To query PROLOG, a new Pengines object is created. Using the attribute `ask`, a query is specified which is handed to the Pengine after creation. Event handlers can be defined for events such as creation and deletion of a Pengine, input and output operations and handling success, failure and exceptions of a query. Here, only a success handler is used. The obtained JSON contains an object `data`, which holds resulting unifications, besides further information about the Pengine. This is converted into plain text and inserted in the document's body. In our case, the following output is given:

```
[{"H":"Hello World"}]
```

After that, the Pengine is automatically destroyed and associated resources are freed after successfully returning the result. However, if specified it is also possible to keep the Pengine alive on the server and perform several queries on it.

Pengines are not limited to our use case. In addition to the communication between PROLOG and JavaScript, they can also be utilised to realise remote procedure calls. These execute a query with the help of multiple PROLOG systems. This allows for requests which involve remote servers, but behave just as a local system [LW14, p. 543].

Listing 4.6: Using Pengines with JavaScript

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Hello Pengines</title>
5    <script src="jquery/jquery-3.2.1.js"></script>
6    <script src="/pengine/pengines.js"></script>
7    <script type="text/x-prolog">
8      hello(H) :- H = 'Hello World'.
9    </script>
10   <script>
11     var pengine = new Pengine({
12       ask: 'hello(H)',
13       onsuccess: function() {
14         $('body').text( JSON.stringify(this.data) )
15       }
16     });
17   </script>
18 </head>
19 <body></body>
20 </html>
```

4.5.2 Built-In Security Features

Since Prolog has full access to the file system, it is important to control what actions can be performed by a remote pengine. The SWI-Prolog library *sandbox* provides the predicate `safe_goal/1`. It evaluates the given goal abstractly and either succeeds or throws an error, if it is not considered safe. To be considered safe, the call-graph must be fully expandable. Unresolvable meta-predicates are not allowed, also every predicate used in the call-graph has to be white-listed. By default, only built-in predicates, which can be considered safe, are white-listed. It is possible to add additional entries utilising the multi-file predicates `safe_primitive/1` and `safe_meta/2`. When utilising Pengines, all goals are validated by `safe_goal/1` automatically prior execution.

Pengines provides a basic protection against *denial of service (DoS)* attacks. The number of concurrently running Pengines can be restricted. In addition, the execution can be aborted after a given time limit and inactive Pengines will be destroyed after a set time.

By providing these features, Pengines suit the usage in web application well. Especially compared to other available approaches, a lot of useful features are already built-in.

5 Approaches to Analyse the DCG Execution

In order to implement the targeted visualisation from Chapter 3, some information related to a grammar has to be gathered. Due to recursive rules, the complete language space of a grammar can easily be infinite. To avoid handling and visualising infinite trees, our approach is not based on only a grammar itself, but on the execution of a query using a DCG. The main goal is to produce a parse tree for a given query. This allows the basic visualisation of the result. For visualising failure, this parse tree needs to be expanded by failing goals. In addition, one way of deriving the order of execution is important to be able to visualise this later accordingly.

There are diverse approaches to analyse the execution of DCGs, several were considered within the scope of this work and are presented here. First, the ability to generate a parse tree using DCGs is discussed in Section 5.1. A quick glimpse on the analysis using other programming languages is given in Section 5.2, followed by an approach using SWI-Prolog's built-in tracer (Section 5.3). Finally, the approach of using a meta-interpreter, which the actual implementation (see Chapter 6) is based on, is presented in Section 5.4.

5.1 Generating Parse Trees with DCGs

It is possible to retrieve a parse tree by extending a DCG. This displays the structure of a difference list which results from a query. Only few modifications have to be done to an already given DCG to achieve this functionality. In our example, the structure of the parse tree should be the following:

- The leave nodes of the tree are holding the terminals.
- The root node and the internal nodes represent the non-terminals.
- The non-terminal nodes should be named after itself, the relationships between the nodes should represent the structure of the DCG.

To achieve this functionality, the grammar rules have to be expanded by an additional argument, which describes the corresponding tree structure.

To realise this, a grammar rule like

```
determiner -> [the].
```

would be expanded to

```
determiner(determiner(the)) -> [the].
```

This modification does not alter the core semantic of this rule, it still only accepts the terminal `the`. The additional argument unifies with the requested tree. In this case, the root node `determiner` with its leaf `the`. Rules with non-terminals, like

```
sentence -> noun_phrase, verb_phrase.
```

can be modified to

```
sentence(sentence(NP,VP)) -> noun_phrase(NP), verb_phrase(VP).
```

Here, the resulting tree has the root `sentence`. The trees from `noun_phrase//1` and `verb_phrase//1` are inserted as its child nodes. The complete modified DCG, based on Listing 4.1, looks like this:

Listing 5.1: Grammar which generates a parse tree

```
1 % structure with parse tree
2 sentence(sentence(NP,VP)) -->  noun_phrase(NP), verb_phrase(VP).
3 noun_phrase(noun_phrase(D,N)) --> determiner(D), noun(N).
4 verb_phrase(verb_phrase(V,NP)) --> verb(V), noun_phrase(NP).
5
6 % vocabulary
7 determiner(determiner(the)) --> [the].
8 determiner(determiner(a)) --> [a].
9 noun(noun(cat)) --> [cat].
10 noun(noun(mouse)) --> [mouse].
11 verb(verb(chases)) --> [chases].
```

When querying this grammar, the added argument provides the proof tree. This works independently of the given input, whether an already defined list or with unbound variables for generating outputs. An example query on this grammar is given below. Additionally to the list `S`, the variable `T` is unified with the corresponding parse tree.

```
?- phrase(sentence(T), S, []).
T = sentence(noun_phrase(determiner(the), noun(cat)), verb_phrase(
      verb(chases), noun_phrase(determiner(the), noun(cat)))),
S = [the, cat, chases, the, cat] .
```

This parse tree can be displayed graphical, as in Figure 5.1. Restricted to the case, that the failing branches should not be displayed, a visualisation using this data could be realised. The sequence of the execution can be read from the tree structure, depth-first, from left to right. An augmentation of the tree is possible in principle for storing additional information. However, this approach has the disadvantage that the DCG has to be modified. Although this can be done automatically via a term-expansion, the changed DCG can no longer be used as the original one if desired.

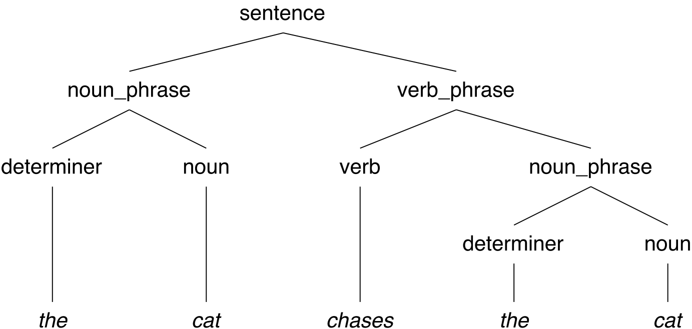


Figure 5.1: Example of a parse tree

DCGs which supply a parse tree are useful when the task is not only to accept or generate lists, but retrieve information about its structure. They do not offer information on backtracking and failed goals, but rather only display the structure at the end of execution, or none if the query failed. However, these intermediate execution steps should be shown in the visualisation. Thus, this approach does not provide sufficient information for the goal of this work.

5.2 Using other Programming Languages than Prolog

Another approach to retrieve information about a DCG is, instead of running a query in an existing Prolog implementation, analysing it in another program environment. This is used in some of the existing research, which was presented in Chapter 2. For instance, *PrettyProlog* [SMM09] utilises an interpreter written in Java, while the *Prolog Visualizer* [LW15] makes use of a JavaScript environment.

In the case of the *Prolog Visualizer*, however, the major drawback of this approach can quickly be recognised: While given sample programs work, the support of built-in ISO predicates is missing. Thus, for example, neither arithmetic evaluation using the predicate `is/2`, nor the use of negation, or predicates which provide direct access to the Prolog database such as `assert/1` or `clause/2` are usable. Also, it lacks support of term-expansions and therefore DCGs are not supported.

Since normal Prolog code can be embedded in DCGs, this problem applies both to the analysis of Prolog and to DCGs. In order to analyse not only simple examples, but also real-world programs, this approach leads to implementing a comprehensive Prolog runtime environment with tracing capabilities from scratch. Since there are already existing Prolog implementations with extensive tracing facilities, this effort is avoidable.

5.3 Using the Built-In Tracer

SWI-Prolog provides, as introduced in Section 4.3, a command line tracer with the possibility to modify its functionality to fit the needs of a specific application. One of the advantages of this approach is that the execution of a query is performed directly by SWI-Prolog. This ensures that the derivation of the result is done correctly without the need of re-implementing any logic. In addition, this offers the advantage that built-in predicates can be evaluated without additional efforts.

5.3.1 Customising the Tracer

The interception of the command line tracer can be accomplished using the predicate `prolog_trace_interception/4`. If declared, each execution step in which the tracer is invoked, is handled by this predicate. This can be used to determine information about the current execution context. If these are stored, it is possible to process them in a subsequent step to retrieve a structured representation of the execution. An excerpt of a custom tracer is given in Listing 5.2.

Listing 5.2: Collecting execution information using the intercepted tracer

```
1 prolog_trace_interception(Port, Frame, Choice, continue) :-
2   get_counter(N),
3   prolog_frame_attribute(Frame, goal, Goal),
4   % ..
5   prolog_frame_attribute(Frame, parent, Parent),
6   % ...
7   prolog_choice_attribute(Choice, frame, ChoiceFrame),
8   % ...
9   assert(step(N, Port, Frame, Goal, ClauseReference, Parent,
      ParentGoal, RecursionLevel, Choice, ChoiceFrame, ChoiceParent,
      ChoiceParentFrame)).
```

Each time the tracer is invoked, a fact `step/13` is asserted. This stores, besides a counter which numbers the individual steps, information about the current frame. These are a reference to the current frame, the associated goal and if the goal has already been unified, a reference to the currently used clause. In addition, a reference to the parent frame and its goal is stored, followed by the recursion level. The last four values provide a reference to the current choice point, its parent choice point, and its frame.

The fourth argument of the predicate `prolog_trace_interception/4` specifies how the execution is continued. Here, the value is `continue`, which equals *creep* in the command line tracer, so every execution step is handled by the tracer. It is also possible to skip the tracing of goals, using `skip`. This is handy, if some goals should be viewed as a black box, for example the execution of built-in predicates.

The usage of this tracer is identical to starting the normal tracer. In this example, the port `unify` was also made visible.

```
?- visible(+unify), trace, phrase(noun_phrase, NP), nodebug.
NP = [the, cat] .
```

Since the command line tracer has been replaced, there is no output to the command line and no user interaction. Only the result and the unifications that have taken place are shown. However, corresponding facts have been asserted, which can now be further analysed. A listing of `step/13` leads to the following output (shortened, variable names and clause references simplified):

```
?- listing(step).
% ...
step(3, unify, 235, noun_phrase(A,[]), <c1>,
     209, $dcg:call_dcg(noun_phrase,A,[]), <c0>, 1,
     246, 235, 121, 105).
step(4,  call, 255, determiner(A,B), _,
     235, noun_phrase(A,[]), <c1>, 2,
     246, 235, 121, 105).
step(5, unify, 255, determiner([the|B],B), <c2>,
     235, noun_phrase([the|B],[]), <c1>, 2,
     266, 255, 121, 105).
step(6,  exit, 255, determiner([the|B],B), <c2>,
     235, noun_phrase([the|B],[]), <c1>, 2,
     266, 255, 121, 105).
% ...
```

This example shows four asserted execution steps: First, the unification of the goal `noun_phrase(A, [])` with a corresponding clause, followed by the execution steps required for satisfying `determiner//0`, which is the first grammar body item in `noun_phrase//0`'s grammar rule body. In the not shown steps at the beginning, `phrase/3` is called, which leads to the call of `$dcg:call_dcg/3`, the parent goal of step 3. The order of the events, which have taken place, can be readout from the first argument: After the unification in step 3, `determiner//0` is called in step 4. Using the reference to the parent's frame and goal, the relationship between both events can be derived. In step 5, the goal is unified to a specific clause, which results in the unification of the goal's arguments with the clause's head. Also, a new choice point is created. The last step shown is the exit of the execution of `determiner//0`. Since this clause does not have any grammar body elements, this step takes place immediately after step 5.

5.3.2 Problems Encountered

In principle, it is possible with this asserted data to obtain comprehensive information on the execution of a query. However, some difficulties arose during the implementation, which is why this approach was not realised.

The main problem encountered is that the number of a frame is no unique identifier. This was assumed in the first place and is important in order to establish a connection between the individual execution steps. Though, there are cases in which the same frame reference is used by several different goals. This has been observed in the context of deterministic goals, for example, in the case of terminals. Probably execution frames which can not generate further solutions are removed from the execution stack and their frame reference is released. However, this is in contrast to the fact that last-call optimisation is disabled in debug mode [Wie17b, p. 37].

Whether this is limited to these observed cases can not be said with certainty. Unfortunately no documentation could be found which contains detailed information on how frames relate to each other, or why frame references are used multiple times. Despite extensive attempts to understand what information can be obtained from the tracer, it was not possible to create a working prototype.

However, it is definitely possible to use `prolog_trace_interception/4` to show the execution of Prolog. The built-in graphical debugger gets the required execution context information using these predicates [Wie17b, p. 75].

5.4 Using Meta-Interpreters

Since it is possible in Prolog to access clauses directly, it is relatively easy to write your own interpreter. An interpreter for the language in which it was written is called a *meta-interpreter (MI)*. Instead of directly calling a goal, this is passed to the meta-interpreter and evaluated.

As explained in Section 4.2.3, SWI-Prolog expands grammar rules to regular Prolog code. Therefore, the meta-interpreters presented here interpret this translated form of DCGs. So these meta-interpreters are not just suited for DCGs. Only slight adjustments are required for the analysis of normal Prolog programs.

In the following, the general idea is to be shown. For sake of simplicity, this is limited to a simple form of rules: The head of a grammar rule is a non-terminal, the body consists of arbitrary terminals and non-terminals. Only conjunction `,/2` is implemented as a control structure. Of course, it is possible to evaluate other allowed structures and components as well, but this would make the meta-interpreters shown here unnecessarily extensive. The DCG which is used for sample queries inside this section was already introduced earlier, see Listing 4.1.

5.4.1 Vanilla Meta-Interpreter

One of the most basic meta-interpreters for pure Prolog is the so-called *Vanilla Meta-Interpreter* [Tri17][SS94, pp. 323–324]. This does not add any additional features, it only allows the execution of a goal. Since terminals of a DCG are translated into unifications, our Vanilla MI is slightly modified to accomplish this.

The MI is amazingly compact. Only a few lines of code are necessary to cover all possible cases: If a goal is literally `true`, it succeeds. Goals which are compound terms using a conjunction succeed if the first and second argument are satisfiable. Unifications should succeed or fail just like in normal Prolog. For all other cases, a grammar rule whose head matches the current goal is determined. This is achieved by using the built-in predicate `clause(Head, Body)`. If a grammar rule was translated into a fact without body elements, `Body` gets unified with `true`, otherwise with its translated grammar body. Subsequently, the body is evaluated. The goal succeeds if its body does. If a goal cannot be satisfied with the current clause, `clause/3` provides alternative clauses on backtracking. Written in Prolog, this can be expressed as in Listing 5.3.

Listing 5.3: Vanilla meta-interpreter suitable for DCGs

```
1 vanilla(true).
2 vanilla((A, B)) :-
3   vanilla(A),
4   vanilla(B).
5 vanilla((A = B)) :-
6   A = B.
7 vanilla(A) :-
8   A \= (_, _),
9   A \= (_ = _),
10  A \= true,
11  clause(A, B),
12  vanilla(B).
```

This interpreter has to be queried with the expanded form of grammar bodies. So the formulation of a query differs compared to the predicate `phrase/3`. Using the arguments added during translation, input and rest lists can be specified. Then, the results are the same as when utilising `phrase/3`. The resulting unifications are as expected and through PROLOG's backtracking features, the Vanilla MI is capable of finding all alternative answers:

```
?- vanilla(sentence(In, [])).
In = [the, cat, chases, the, cat] ;
In = [the, cat, chases, the, mouse] ;
% ...
```

Of course, this MI has no practical benefit, except showing off the concept. However, it can be adjusted to fit specific needs. For instance, it is possible to replace the standard search strategy of PROLOG with a MI. Handling left-recursive grammars or limiting the depth of the search tree are only two examples, which can be achieved with modified MIs [Tri17]. Another possibility is to use MIs to output information about the execution. Especially the latter is relevant in our case, the analysis of the DCG execution.

5.4.2 Generating a Parse Tree

Comparable with the expansion of a DCG by an additional parameter for outputting a parse tree (see Section 5.1), it is also possible to adapt a MI accordingly. This has the advantage that the DCG has not to be modified while the MI can be used for arbitrary DCGs. A simple customised MI is given in Listing 5.4.

Listing 5.4: Meta-interpreter with tree output

```
1 mi_proof(true, true).
2 mi_proof((A, B), (TA, TB)) :-
3   mi_proof(A, TA),
4   mi_proof(B, TB).
5 mi_proof((A = B), (A = B)) :-
6   A = B.
7 mi_proof(A,(A:T)) :-
8   A \= (_, _),
9   A \= (_ = _),
10  A \= true,
11  clause(A, B),
12  mi_proof(B, T).
```

This meta-interpreter succeeds when a solution for the query has been found and the second argument has been unified with the parse tree. The representation used here slightly differs from the modified DCG in Section 5.1. For example, the parent-child node relationship is expressed using the operator `:/2`. On the left side, there is the parent goal, on the right, a comma separated list of its child nodes. Also, the terminals are provided differently as in the structure generated by the DCG. By determining the difference between the input and remainder lists of a non-terminal these can be read out. In this representation, only a rule's arguments and no further tree traversal is needed for determining the list it represents.

When querying this MI, additionally to the expected result, `T` is unified with the parse tree which could be further examined:

```
?- mi_proof(verb_phrase(In, []), T).
  In = [chases, the, cat],
  T = verb_phrase([chases, the, cat], []):(
        verb([chases, the, cat], [the, cat]):true,
        noun_phrase([the, cat], []):(
          determiner([the, cat], [cat]):true,
          noun([cat], []):true
        )
      ) ;
   % ...
```

However, this MI does not extend the information which is gathered during the execution compared to the modified DCGs from Section 5.1. The biggest drawback

with respect to our use case is still, that failing execution branches are neglected during backtracking and therefore not represented in the tree.

5.4.3 Picture Backtracking

Both the Vanilla MI and the MI which displays a parse tree, use PROLOGs built-in backtracking mechanisms. This is obvious since the functionality is already given and easily implementable, but results in discarding gathered information when back-tracking occurs.

To picture backtracking, one approach is to re-implement its logic. For instance, this was done in a meta-interpreter by D. Bowen [Bow84] for generating AND/OR trees. This approach requires far-reaching changes to our simple vanilla meta-interpreter. Not only have already gathered data and pending goals to be passed within the MI, even basic mechanisms such as unification can no longer be utilised. Instantiated variables can be made unbound during backtracking and re-instantiated later, dependent on the current choice. This can not be easily achieved with normal variables. In the case of the AND/OR meta-interpreter, uninstantiated variables get converted into terms using `numbervars/3`. These are then used to associate actual values to a variable. This is both complex as well as error-prone and generates many additional computing steps.

Another approach is to store information during execution in a way that it is not affected by backtracking. This has the advantage that PROLOG functionality, such as backtracking and unification, can be used as usual. In addition to writing to a stream like the command line, storing facts in dynamic predicates or the recorded database is a possibility. These actions are not reverted during backtracking. In this way, the MI can utilise backtracking and unification mechanisms, and still obtain information about failed execution steps. This kind of meta-interpreter can be used to build customised tracers [Bra12, pp 609–610].

5.4.4 Tracing Meta-Interpreter

In the following, a meta-interpreter is presented, which also displays failing rules. This is achieved by writing statements to the command line during execution. The tracing meta-interpreter is given in Listing 5.6.

Compared to the Vanilla MI, it is modified in various ways: Extended by one argument, the current recursion level is stored for an indentation while writing to the command line [SS94, p. 328]. Operations which could fail but are of interest,

like the unification of two values, are executed inside the condition of an If-Then-Else statement. Thus, in the case of a failure, the Else-branch is executed instead of backtracking. This can be used to output a message of failure to the command line, followed by an explicit invocation of backtracking using the built-in predicate `fail/0`. With the help of the built-in `nth_clause/3` a candidate clause, with an index number and unique signature, is found. To determine a clause's corresponding body, `clause/3` is used. After incrementing the recursion level, it is evaluated.

To express an If-Then-Else statement, SWI-Prolog offers several options: The most obvious choice is the operator `->/2`. However, utilising this operator, if a solution for the condition has been found, choice points for it are destroyed. Thus, this does not fit our needs. As an alternative, there is also the operator `*->/2` in SWI-Prolog, which implements a soft-cut. With this operator, Prolog will backtrack over multiple solutions of the condition. The Else-branch is only called if the condition cannot be satisfied at all. If a solution was found and then additional are searched via backtracking, the Else-branch will not be executed in case of failure. In our case, the Else-branch should also be called in these cases for displaying these failing execution steps. For this, the operator `~>/2` (see Listing 5.5) has been defined, which provides the required functionality.

Listing 5.5: Handling If-Then-Else statements

```
1 :- op(1150, xfy, ~>).
2 Goal ~> Then ; _Else :- Goal, Then.
3 _Goal ~> _Then ; Else :- !, Else.
```

Listing 5.6: Meta-interpreter with output of failure

```
1  mi_print(true, _Level).
2  mi_print((A, B), L) :-
3    mi_print(A, L), mi_print(B, L).
4  mi_print((A = B), L) :-
5    (A = B ~>
6      print_indented((A = B):exit, L)
7    ;
8      print_indented((A = B):fail, L),
9      fail
10   ).
11 mi_print(A, L) :-
12   A \= (_,_), A \= (_=_), A \= true,
13   print_indented(A:call, L),
14   nth_clause(A, Index, Sign),
15   L1 is L + 1,
```

```
16   (clause(A, B, Sign), mi_print(B, L1) ~>
17     print_indented(A:Index:exit, L)
18   ;
19     print_indented(A:Index:fail, L),
20     fail
21   ).
22
23 print_indented(A, L) :-
24   Indent is 2*L, tab(Indent), writeln(A).
```

When querying a DCG, the resulting structure is displayed as a sideways tree, child elements are further indented. The single lines use the format `<goal>(:<index>):<call | exit | fail>`, respectively in the case of unifications `<unification>:<exit | fail>`. Failing clauses are also printed, before calling available alternatives. Thus, output is also possible if a query fails, as shown in the example below:

```
?- mi_print(verb_phrase([chases, a, dog], []), 0).
verb_phrase([chases,a,dog],[]):call
  verb([chases,a,dog],_6966):call
  verb([chases,a,dog],[a,dog]):1:exit
  noun_phrase([a,dog],[]):call
    determiner([a,dog],_7198):call
    determiner([a,dog],_7198):1:fail
    determiner([a,dog],[dog]):2:exit
    noun([dog],[]):call
    noun([dog],[]):1:fail
    noun([dog],[]):2:fail
    determiner([a,dog],_7198):2:fail
  noun_phrase([a,dog],[]):1:fail
  verb([chases,a,dog],_6966):1:fail
verb_phrase([chases,a,dog],[]):1:fail
false.
```

Similar to the command line tracer, each performed execution step is printed. This meta-interpreter performs the following execution steps, while trying to satisfy the goal: The predicate `verb_phrase//0` has only one clause, its grammar body is formed by a `verb//0`, followed by a `noun_phrase//0`. To prove the goal, first `verb_phrase//0` is called and a matching `verb//0` for its first grammar body item is found. Since SWI-Prolog has moved the unification of the terminal into the rule's

head during the translation of the grammar, this is done directly without further unification steps. Then it is tried to describe the remaining list with a `noun_phrase//0`. The second clause of `determiner//0` fits the query. However, since `[dog]` is no valid `noun//0` in the scope of our grammar, this fails. Then, the meta-interpreter tries to find alternatives using backtracking. Thus, previously succeeding goals are retried. Since neither `determiner//0`, `verb//0`, nor `verb_phrase//0` have further open choice points, these fail and the correct result, *false*, is returned.

If, instead of writing to the command line, individual execution steps are stored as facts, it is possible to generate an augmented parse tree with failing branches from these in a further step. In order to be able to assign the facts later correctly, an appropriate way of identifying the nodes has to be added. For example, when backtracking occurred, it is currently not always clear what the parent of a goal is. The carried recursion level and the numbering of alternative clauses is only a first direction. The extended MI, which is used in the implementation, is presented in-detail in Chapter 6.

6 DCG Tracer

A tracing meta-interpreter for DCGs in SWI-Prolog is introduced in this chapter. This provides the necessary information which is required for the visualisation.

6.1 Concept

The way the program works is structured as shown in Figure 6.1: A given DCG is first translated into normal Prolog code with the help of a modified term-expansion. Subsequently, this can be used for queries. These are evaluated by the meta-interpreter, information about each relevant execution step is temporarily stored as individual facts. Then, these are summarised in a dict, which represents the entire execution.

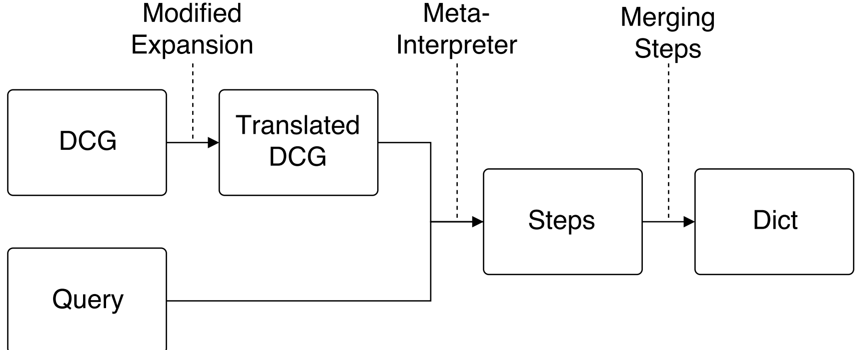


Figure 6.1: Overview of the structure of DCG Tracer

6.2 Modified DCG Translation

As explained in Section 4.2.3, SWI-Prolog translates DCGs into normal Prolog code and performs some optimisations on them. For instance, if the first grammar body item is a terminal, it is moved inside the translated rule's head. Even if this does not change the result of a query, it changes which clauses are invoked during execution [Wie17b, p. 58]. It is to be assumed that this optimisation can lead to confusion when visualising an execution since some rules which fail immediately are not even invoked and therefore not displayed.

Within the built-in predicate `dcg_translate_rule/2`[10], these optimisations are executed as a final translation step. Since user-defined term-expansions are executed before SWI-Prolog's DCG translation, it is possible to bypass any optimisations by directly calling the system predicates used for translating DCGs, excluding the final optimisations.

By default, terminals are translated into a unification of lists. After executing unifications on them, it is not possible to directly distinguish the input, rest and terminal list any more. To fix this, we add an additional wrapper. Each terminal is replaced by a call of the non-terminal `'T'//1`, with the terminal as argument. Thus, by the translation in normal Prolog code, this becomes a call of the predicate `'T'/3`.[11] This predicate performs the unification between the terminal, input and rest list. These three lists are clearly separated from each other while providing the expected behaviour. The definition of `'T'/3` is given in Listing 6.1.

Listing 6.1: Definition of `'T'/3`

```
1 'T'(Terminal, In, Rest) :-
2   append(Terminal, Rest, In).
```

Similarly, normal Prolog code in grammar bodys, which is nested between curly brackets `{..}`, can be wrapped. For this purpose, the meta-predicate `'P'/3` is defined which calls its first parameter and unifies the handled input and rest lists. This allows the distinction between non-terminals and normal Prolog code inside a grammar body. Depending on whether the normal Prolog code is to be handled as a black box or not, the wrapper can either be applied to the complete content inside the brackets at once or to each individual element. The implementation treats nested Prolog code as a black box, showing only the result while hiding performed execution steps. Cuts `!/0` can not be used at the moment. If these are to be interpreted, it should be noted that these must not be wrapped inside `'P'//1` since meta-calls are opaque to the cut [Wie17b, p. 108]. Otherwise the cut would only have an effect within the predicate `'P'/3`, but not in the corresponding grammar rule.

To illustrate the differences between the normal DCG translation by SWI-Prolog and this modified translation, a DCG for palindromes, consisting of the letters `a,b,c`, is given in Listing 6.2.

[10] See `dcg_translate_rule/2` in module `'$dcg'`: `http://www.swi-prolog.org/pldoc/doc/_SWI_/boot/dcg.pl?show=src#dcg_translate_rule/2`.

[11] This should not be confused with the predicate `'C'/3`, which is used in some implementations for expressing terminals. Although the use case is similar, the order of arguments differs [CM03, 289–294].

Listing 6.2: DCG for palindromes

```
1 palin --> [].
2 palin --> letter(_).
3 palin --> letter(X),
4           palin,
5           letter(X).
6 letter(X) --> [X],
7           { member(X, [a,b,c]) }.
```

By default, this DCG is translated by the built-in predicates into the following clauses:

```
% SWI's default translation of grammar rules
?- listing(palin), listing(letter).
palin(A, A).
palin(A, B) :- letter(_, A, B).
palin(A, E) :-
  letter(C, A, B),
  palin(B, D),
  letter(C, D, E).
letter(A, [A|C], B) :-
  member(A, [a, b, c]),
  B=C.
```

During the translation of the first rule, the empty terminal was moved in the clause's head, thus resulting in an empty clause's body. This is also the case with the clause corresponding to `letter//1`. In addition, the nested Prolog code is inserted, followed by the unification of the input and rest list. It is not clear any more, whether `member/2` is a translated non-terminal, or results from normal Prolog code inside a grammar rule. Our modified DCG translation provides clauses which look closer to the original grammar rules:

```
% Modified translation of grammar rules
?- listing(palin), listing(letter).
palin(A, B) :- 'T'([], A, B).
palin(A, B) :- letter(_, A, B).
palin(A, E) :-
  letter(C, A, B),
  palin(B, D),
  letter(C, D, E).
```

```
letter(A, B, D) :-
  'T'([A], B, C),
  'P'(member(A, [a, b, c]), C, D).
```

Now each grammar body item is still clearly identifiable as such a one. Also, both terminals and PROLOG code are in the same place as in the grammar rules. No unifications are moved inside a clause's head. Querying the DCG using the built-in predicates like `phrase/3` is still possible, since the predicates `'T'/3` and `'P'/3` offer the required functionality. Of course, this program performs some additional computation steps, since additional calls are necessary and rule indexing is bypassed. However, the executed steps are closer related to the initial formulated DCG. Since DCGS, not the expanded rules, are to be visualised, this is a desirable property of the translated clauses.

6.3 Tracing Meta-Interpreter

The provided meta-interpreter is based on the idea presented in Section 5.4.4. Handled code is executed, while gathering information about the execution. This approach is modified to match the application needs:

- **Handling of the modified translation of DCGs**
  Instead of handling explicit unification, the MI evaluates the predicate `'T'/3`. Furthermore, nested PROLOG code is handled by the predicate `'P'/3`.
- **Visible Ports**
  This meta-interpreter traces the ports `unify`, `exit` and `fail`.
- **Asserting Facts**
  Each execution step is asserted as a fact. Due to the stored context, a distinct assignment of related steps to one another is possible. An overview of the structure and an example of these facts is given in Section 6.4
- **Numbering execution steps**
  Each step has a unique identifier, which also represents the order of execution.
- **Depth limit**
  A depth limit has been added to intercept too extensive queries and left-hand recursion. This is represented by the additional event `depth`.

The meta-interpreter is given in Listing 6.3. Further predicates are used in the code which are not shown in more detail. On the one hand, this is the predicate `save_step/9`. This evaluates the given arguments and saves them as a fact.

The first two arguments are optional, the first one numbers execution steps, the second argument is the value of a redo counter, which indicates how often backtracking was used so far. The other predicate is `current_counter/3` which increments a counter value. In this section, only the meta-interpreter is shown. An example of the resulting facts when using this meta-interpreter is given in the subsequent Section 6.4.

The actual interpreter consists of four clauses: `mi/1` serves for initialisation purposes and calls the actual interpretation. The three clauses of `mi/2` evaluate a goal, generate the calls to store events, and call subgoals.

Initialisation

The MI is called with the predicate `mi/1`. Instead of a DCGBody, as is the case of the built-in `phrase/3`, a DCGBody which was translated into normal PROLOG is expected as an argument. As mentioned before, this translated form also contains the input and rest lists, so these can be instantiated.

First, a step with the type *unify* is stored, which represents the root of the query. Then the actual MI is called. As introduced in Section 5.4.4, this is done inside a custom If-Then-Else statement. `mi/2` requires two arguments: the current goal and a compound. The latter specifies the recursion level, a reference to the parent goal, a numbering of its subgoals, and a reference to the module which holds the called DCG. If the request is successful, an event *exit* is saved, otherwise an event *fail*. Finally, the counter *redo* is increased, in order to provide the correct counter value if backtracking is triggered to find further solutions.

Handling Conjunction

Conjunctions are treated in a similar way as in the meta-interpreters already presented in Section 5.4. Here, the second goal is called with an incremented value of the in-clause position `Pos`.

Handling Terminals and Prolog Code

Since the modified DCGS from Section 6.2 are interpreted, no unifications using the predicate `=/2` are present. Instead, calls of `'T'/3` for terminals and `'P'/3` for PROLOG code have to be evaluated.

First, it is checked for a given goal if it has a corresponding functor and arity. If this succeeds, a step *unify* is stored and the goal is evaluated. After that, the result of

the goal is stored. If the goal does not succeed, the counter redo is increased and backtracking is invoked by calling `fail/0`.

Handling Non-Terminals

Non-terminals are handled by the third clause of `mi/2`. Using `clause/3`, a matching clause, with its body and a reference to this clause, is determined. Subsequently, this unification, with a reference to the current clause, is stored.

The recursion level is incremented, then the body is executed by the meta-interpreter and the result is stored. The value of `step` from the previous unification of this clause is used as the parent step for this call.

Due to the modified DCG translation, there are no rules that result in facts. Therefore, a rule that evaluates an empty clause's body is not needed, compared to the meta-interpreters from Section 5.4.

Depth-Limiting

The predicate `save_step/9` (not included in the listing) checks the reached recursion level prior to the assertion of a step. If a predefined limit has been reached, the step is stored as an event *depth*, followed by a call of `fail/0`, to stop further execution. This ensures that executions which reached the maximum depth are terminated. This intercepts both left-recursive calls and too extensive requests.

Listing 6.3: Tracing meta-interpreter for DCGs

```
1 % Initialisation
2 mi(Module:Expanded) :-
3   save_step(unify, N, 1, 0, -1, 1, Expanded, 'DCGBody', Module),
4   (mi(Expanded, 1:N:1:Module) ->
5     save_step(exit, _, _, 0, -1, 1, Expanded, 'DCGBody', Module)
6   ;
7     save_step(fail, _, _, 0, -1, 1, Expanded, 'DCGBody', Module),
8     fail
9   ),
10  update_counter(redo, _, Module).
11 % Conjunction
12 mi((A, B), Lvl:Par:Pos:Module) :-
13   Pos1 is Pos + 1,
14   mi(A, Lvl:Par:Pos:Module), mi(B, Lvl:Par:Pos1:Module).
15 % Terminals & Prolog Code
16 mi(A, Lvl:Par:Pos:Module) :-
17   functor(A, F, 3), (F = 'T' ; F = 'P'),
18   save_step(unify, _, _, Lvl, Par, Pos, A, F, Module),
19   (call(A) ->
20     save_step(exit, _, _, Lvl, Par, Pos, A, F, Module)
21   ;
22     save_step(fail, _, _, Lvl, Par, Pos, A, F, Module),
23     update_counter(redo, _, Module),
24     fail
25   ).
26 % Non-Terminals
27 mi(A, Lvl:Par:Pos:Module) :-
28   A \= (_,_), A \= 'T'(_,_,_), A \= 'P'(_,_,_),
29   clause(Module:A, B, ClauseRef),
30   save_step(unify, N0, _, Lvl, Par, Pos, A, ClauseRef, Module),
31   L1 is L + 1,
32   (mi(B, L1:N0:1:Module) ->
33     save_step(exit, _, _, Lvl, Par, Pos, A, ClauseRef, Module)
34   ;
35     save_step(fail, _, _, Lvl, Par, Pos, A, ClauseRef, Module),
36     update_counter(redo, _, Module),
37     fail
38   ).
```

6.4 Representing the Execution as Facts

During the execution of a query, facts are asserted to the Prolog database. Each fact represents a single event in the 6-port execution model (see Section 4.3.1). The ports `unify, exit` and `fail` are traced. Instead of displaying the port `redo` like in the command line tracer, it is replaced by a counter which holds the number of occurred redos. The port `call` is skipped in favour of the port `unify`. This port is enabled when a clause is found which matches a goal. Since the clauses' heads do not contain unifications of terminals due to the modified DCG translation, this port differs from the port call only by the selection of a corresponding clause. Exceptions are not further handled by the interpreter. In addition to the visible events from the 6-port model, there is the event `depth` which marks the aborted execution of goals due to a previously reached depth limit.

Each step is represented by a single fact `step/8` with the following arguments:

- **Port**
  Stored event, this is a member of *unify, exit, fail* and *depth.*

- **Step**
  Current execution step. This is used both for determining the sequence of execution as well as uniquely identify each event.

- **Redo**
  Counts the redos that occurred prior this step. Each time when a execution step fails and alternative solutions are searched using backtracking as well as when requesting further solutions, this value is increased.

- **Level**
  The recursion level of the event. The toplevel has the level 0, calls within a rule's body lead to an incremented recursion level.

- **Parent**
  A reference to the step of its parent goal.

- **Position**
  For each grammar rule, grammar body items are numbered. The *position* represents the in-term position of the current goal inside its parent goal.

- **Goal**
  A snapshot of the goal with its current variable bindings. Since uninstantiated variables are no longer distinguishable when they are stored as a fact, the goal is converted to a string beforehand.

- **ClauseRef**
  After calling a goal, it is unified with the head of an appropriate clause. In SWI-Prolog, each clause has a reference which can be obtained using the predicates `clause/3` or `nth_clause/3`. For each non-terminal, the candidate clause used in the current step is referenced. A `'T'` respectively `'P'` marks terminals and embedded Prolog code. The toplevel query, which is not a non-terminal but a grammar body, gets the value `'DCGBody'` assigned.

When using the meta-interpreter, steps are asserted by the predicate `save_step/9`. Except for *step* and *redo*, argument values are provided by the meta-interpreter. In addition, the module in which the fact is to be stored is stated as a parameter.[12] To count the execution steps and redos, an interface for simple counters is provided. For each named counter, a dynamic fact `counter/2` is created which holds the current value. These counter values are not affected by backtracking.

As an example, using the meta-interpreter with a sample query results in the following facts (variable names and clause references simplified):

```
?- mi(noun_phrase(A,[])), listing(step).
% step(Port, Step, Redo, Lvl, Par, Pos, Goal, ClauseRef)
step(unify, 1, 1, 0,-1, 1, "noun_phrase(A, [])", 'DCGBody').
step(unify, 2, 1, 1, 1, 1, "noun_phrase(A, [])", <c1>).
step(unify, 3, 1, 2, 2, 1, "determiner(A, B)", <c2>).
step(unify, 4, 1, 3, 3, 1, "'T'([the], A, B)", 'T').
step(exit,  5, 1, 3, 3, 1, "'T'([the], [the|B], B)", 'T').
step(exit,  6, 1, 2, 2, 1, "determiner([the|B], B)", <c2>).
step(unify, 7, 1, 2, 2, 2, "noun(B, [])", <c3>).
step(unify, 8, 1, 3, 7, 1, "'T'([cat], B, [])", 'T').
step(exit,  9, 1, 3, 7, 1, "'T'([cat], [cat], [])", 'T').
step(exit, 10, 1, 2, 2, 2, "noun([cat], [])", <c3>).
step(exit, 11, 1, 1, 1, 1, "noun_phrase([the, cat], [])", <c1>).
step(exit, 12, 1, 0,-1, 1, "noun_phrase([the, cat], [])", 'DCGBody').
A = [the, cat] ;
step(fail, 13, 2, 3, 7, 1, "'T'([cat], B, [])", 'T').
step(fail, 14, 3, 2, 2, 2, "noun(B, [])", <c3>).
step(unify,15, 4, 2, 2, 2, "noun(B, [])", <c4>).
step(unify,16, 4, 3,15, 1, "'T'([mouse], B, [])", 'T').
step(exit, 17, 4, 3,15, 1, "'T'([mouse], [mouse], [])", 'T').
```

[12] This allows for asserting the facts inside the current user's Pengine. Since a Pengine is destroyed when not needed anymore or after a given time limit, this prevents the accumulation of unnecessary facts inside the meta-interpreter's module. Also it enables concurrent requests to the module by multiple Pengines.

```
step(exit, 18, 4, 2, 2, 2, "noun([mouse], [])", <c4>).
step(exit, 19, 4, 1, 1, 1, "noun_phrase([the, mouse],[])", <c1>).
step(exit, 20, 4, 0,-1, 1, "noun_phrase([the, mouse],[])",'DCGBody').
A = [the, mouse] .
```

The example utilises the DCG from Listing 4.1. The first step initialises the execution and represents the query. Here, this is solely the non-terminal `noun_phrase//0`. However, an arbitrary translated DCG body could also be queried instead. The recursion level of this step is `0`, and it has no parent goal, indicated by `-1`. Then, the first, and in our case also the only, grammar body item is executed. The queried `noun_phrase//0` has two subgoals: `determiner//0` and `noun//0`. The events that occur for each goal are displayed. For instance, after invoking `noun_phrase//0`, `determiner//0` is unified with its first clause in step 3. After finding a solution for its subgoal in steps 4–5, which is only a single terminal, the exit port is enabled in step 6. The relationship between these two goals can be seen from the arguments `Step`, `Parent` and `Position`: The goal in step 4 is the first subgoal of step 3. Also, the recursion level is increased by one. The exit of `determiner//0` leads to calling the next subgoal of `noun_phrase//0`, `noun//0`, whose unification is executed in step 7. This is followed by the execution of its subgoal in step 8–9. The execution finishes with exiting the queried DCG body.

Since the given query does not contain any failing goals, the redo counter value is `1` up to step 12. In the case of backtracking, each attempt to find an alternative leads to an incrementation of this value. This can be observed when searching for the next solution of `noun_phrase//0`. Since there are no other choices for unifying the terminal or inside the current clause of `noun//0`, both fail in steps `13-14`. Then an alternative clause for `noun//0` is determined, this contains the terminal `[mouse]`, which leads to the second solution of the query.

In principle, this data is comparable to what the tracer, presented in Section 5.4, provides. However, here it is ensured that distinct assignments of the steps to one another are possible. Using the *Parent* and *Position* arguments as a key, each goal in the box model is uniquely identifiable. Subgoals refer to the corresponding unification step of its parent goal. Thus, a clear distinction of alternatives is possible.

6.5 Representing the Execution as a Dict

After creating individual facts by the meta-interpreter, they are summarised in a single nested structure. The structure is roughly based on the Byrd Box model. Associated events are stored for each goal. Also, subgoals are nested inside the parent

goal. A dict is used as data structure. This represents events which occurred during execution and serves as an exchange format between SWI-Prolog and JavaScript. This works because dicts are converted to JSON by Pengines. First, the structure of the dict is presented, followed by an brief insight into how it is generated.

Each execution step, stored as fact `step/8`, can be represented as a dict `event`. As attributes this contains the execution step number, the type of the event, and the current redo counter value. In addition, the goal is stored with its current variable bindings. Facts `step/8` contain a clause reference. This is used to determine the associated line number of the rule from the DCG used. The built-in predicate `clause_property/2` is utilised for retrieving this information. If there is no meaningful line number, as in the case of terminals, an identifier for the type of the goal is provided.

```
event{
  step: <Step>,
  type: <unify | exit | fail | depth>,
  redo: <Redo>,
  goal: <Goal with current variable bindings>,
  line: <Line number of associated clause | 'T' | 'P' | 'DCGBody'>
}
```

All events associated with a goal are summarised in a dict `goal`. In addition to the list of events, this includes a list of choices. For each unification of a goal, executed subgoals are stored in a separate choice.

```
goal{
  events: <List of event>,
  choices: <List of choice>
}
```

Each dict `choice` consists of a list of its subgoals and a reference to the execution step which lead to this choice.

```
choice{
  parent: <Step>,
  subgoals: <List of goal>
}
```

The result and the steps taken can thus be represented as a nested structure. The root is a dict `goal`, containing the queried DCG body. The executed subgoals and execution steps are nested inside. An excerpt from the dict which results from the example in Section 6.4, is intended to illustrate the structure. Shown is only the dict of the goal `noun//0`, which is part of the complete resulting dict. To reduce the size of this example, the attributes `line` and `redo` are not printed for every event.

```
% ..
goal{
  events:[
    event{goal:noun(B, []), step:7, type:unify, line:6, redo:1},
    event{goal:noun([cat], []), step:10, type:exit, ..},
    event{goal:noun(B, []), step:14, type:fail, ..},
    event{goal:noun(B, []), step:15, type:unify, line:7, ..},
    event{goal:noun([mouse], []), step:18, type:exit, ..}],
  choices:[
    choice{parent:7, subgoals:[
      goal{
        events:[
          event{goal:'T'([cat], B, []), step:8, type:unify, ..},
          event{goal:'T'([cat], [cat], []), step:9, type:exit},
          event{goal:'T'([cat], B, []), step:13, type:fail}],
        choices:[choice{parent:8, subgoals:[]}]
      }
    ]},
    choice{parent:15, subgoals:[
      goal{
        events:[
          event{goal:'T'([mouse], B, []), step:16, type:unify},
          event{goal:'T'([mouse], [mouse], []), step:17, type:exit}],
        choices:[choice{parent:16, subgoals:[]}]
      }
    ]}
  ]
}
% ..
```

The goal `noun//0` has five events. These describe the unification with its first clause, the rejection of this first solution, indicated by the event `fail` in step `14`, and finding the second clause and its solution. This leads to two choices, one for each unification.

Both include the subgoals of the corresponding clauses. In both cases, this is just a terminal as a subgoal, so these have no further subgoals. Therefore, only the associated events and a choice without any subgoals are stored there.

There are three predicates which are used to create the dict: `goal/3` requires the module and the unique identifier of a goal, the latter is composed of the parent step and position. The corresponding dict is unified with the third argument. First, the events associated with the goal are determined using the predicate `event/3`. With the help of the built-in predicate `findall/3`, all corresponding events are found. Every unification of a goal leads to a choice. These are evaluated in a second step by the predicate `choice/3`, which identifies the subgoals and calls `goal/3` on each, providing a resulting dict of type `choice`. Thus, the corresponding dict is created recursively for the given top level goal.

6.6 Provided Modules

The PROLOG code which implements *DCG Tracer* is split in three modules. First, there is the module *dcg_term_expansion*. This performs the modified DCG translation as introduced in Section 6.2. Second, the module *dcg_tracer* provides predicates for querying a DCG and retrieving the dict. In addition, the modified version of the SWI-PROLOG system library `'$dcg'` is provided.

Module: dcg_term_expansion

This module serves to modify the DCG translation, as presented in Section 6.2.

- **dcg_expansion(+DCGRule, -Expanded)**
  For a given rule, its expanded form is provided. This is a replacement for the built-in `dcg_translate_rule/2`. Optimisations are disabled, terminals and PROLOG code are wrapped in the predicate `'T'/3`, respective `'P'/3`. Since term-expansions are not exported beyond module boundaries, it has to be added manually as a term-expansion in the module that holds a DCG.

- **'T'(?Terminal, ?Input, ?Rest)**
  Succeeds, if `Terminal` is the difference between the lists `Input` and `Rest`.

- **'P'(+PrologCode, ?Input, ?Rest)**
  Handled `PrologCode` is called. If it succeeds, the lists `Input` and `Rest` are unified. This is restricted to calling built-in predicates.

Module: dcg_tracer

This module provides an interface to query DCGs and to get a corresponding dict. The initialisation, creation and clean-up of `step/8` and `counter/2` facts is encapsulated by these exported predicates.

- **phrase_mi(:DCGBody, ?Input, -Dict)**
  The same as `phrase_mi(:DCGBody, ?Input, [], -Dict)`.
- **phrase_mi(:DCGBody, ?Input, ?Rest, -Dict)**
  Analogous to the built-in `phrase/3`, solutions for a given `DCGBody` are calculated. Further solutions can be retrieved via backtracking. While the handled `:DCGBody` is extended inside this rule, the queried DCG has to be translated using `dcg_term_expansion:dcg_expansion/2` beforehand. In addition, a dict is unified that represents all executed steps. If a query fails, `Input` and `Rest` will not be unified with a solution. A dict which displays the execution steps which lead to failure is unified nevertheless. This predicate uses the built-in `setup_call_cleanup/3` to retract asserted facts after finishing the goal.
- **phrase_mi_nth(:DCGBody, ?Input, +Nth, -Dict)**
  The same as `phrase_mi_nth(:DCGBody, ?Input, [], +Nth, -Dict)`.
- **phrase_mi_nth(:DCGBody, ?Input, ?Rest, +Nth, -Dict)**
  Similar to `phrase_mi/4`, a dict is unified in addition to the result of a query. Instead of creating several solutions using backtracking one after another, the parameter `Nth` is used to specify the number of the solution to be output. Here, asserted facts are also deleted after finishing the query. The built-in `findnsols/4` is used to find the `Nth` solution. In the case of a failing query, a dict is also provided here. However, if it is not possible to calculate `Nth` solutions, this predicate fails.

Module: swi_dcg

The SWI-Prolog system library `'$dcg'` handles DCG translation and provides also predicates for querying DCGs. The module `swi_dcg` is a modified copy which exports only one predicate for translating grammar rules without optimisations or caching of translated grammar bodies. This is used internally by `dcg_term_expansion:dcg_expansion/2`.

- **dcg_translate_rule_(+Rule, -Clause)**
  Behaves like the built-in predicate `dcg_translate_rule/2`, except for the waiving of optimisations.

The developed modules are independent of the web application and Pengines. Accordingly, it is possible to use these also for other purposes. Unfortunately, these modules will not work in other Prolog implementations. Used language features and built-ins such as dicts, `clause/3` or `findnsols/4` are not part of ISO Prolog, but SWI-Prolog specific.

6.7 Future Work

The current implementation already covers the main part of the language features of DCGs as introduced in Section 4.2.2. Thus, extensive analyses of DCGs are possible. However, there are still possibilities for expansion.

Nested Prolog code in curly brackets `{..}` may currently only use built-in predicates. This provides a rich set of predicates and should be sufficient for most use cases. The definition of additional clauses and their usage within the DCG is currently not possible. Furthermore, the expansion and interpretation of semicontext is not implemented yet.

The meta-interpreter expects clauses whose bodies only contain grammar body items or conjunctions of those. Disjunctions can already be expressed by defining several rules per non-terminal indicator. Adding the interpretation of further control structures in grammar bodies is possible with the provided approach. This can be done analogously to the interpretation of conjunctions by using Prolog's control structures. The cut is most likely to be the most complicated control structure to implement because it must remove choice points not within goals of the meta-interpreter but with respect to the executed query. There are already approaches for implementing cuts in meta-interpreters which could be used as a starting point [Imp14][SS86, 331–332].

In addition to the execution of these control structures, an expansion of the generated dict would also be reasonable then: Instead of only listing subgoals inside each dict `choice`, the structure of the clause could also be represented. Since the clause's body is accessible using `clause/3`, this information can be retrieved directly when a clause is unified.

Currently, an overview of occurred unifications is only indirectly available by comparing the stored goals of several events. An extension of an explicit overview of those would be conceivable. Also, the naming of unbound variables could be further improved. These are named starting with an underscore, followed by a number, as usual in SWI-Prolog.[13]

[13] The latter is already solved by dynamically renaming variables in the visualisation.

7 DCG Visualiser

The application *DCG Visualiser* uses the introduced meta-interpreter *DCG Tracer* to analyse and visualise the execution of querys on DCGs. First of all, the implementation is discussed in Section 7.1, followed by an overview of provided features in Section 7.2.

7.1 Implementation

The visualisation is realised as a web application. It consists of a server that provides HTTP functionality and manages Pengines. On the client side is the web application, where requests are formulated and received results visualised accordingly.

7.1.1 Server

The server is based on the demo server by the Pengines project.[14] This is adjusted accordingly to provide features needed for the application. The web server is already preconfigured. This uses SWI-Prolog's library *http* for handling HTTP requests and responses. In addition, the library *Pengines* is loaded. Unnecessary sample applications have been removed from the demo server.

Since the basic functionality of the server is already given, only minor additions are necessary: The previously introduced *DCG Tracer* must be made available for the usage within Pengines. For this purpose, modules presented in Section 6.6 are loaded. Afterwards, they are only available for the server, but not for Pengines running on it. With the use of `sandbox:safe_meta/2` and `sandbox:safe_primitive/1`, required predicates are also made accessible for them.

A term-expansion calling the predicate `dcg_term_expansion:dcg_expansion/2` ensures that DCGs loaded within a Pengine are properly translated. In addition, to use the meta-interpreter, the predicate `dcg_tracer:phrase_mi_nth/5` is declared as a safe meta-predicate.

[14] A Prolog Engine application server: `https://github.com/SWI-Prolog/pengines`.

To prevent the execution of PROLOG code which should be constrained, like access to the file system or the termination of PROLOG, in nested PROLOG code in grammar bodies, it is checked prior calling. This is executed in the predicate `dcg_term_expansion:'P'/3` by means of `sandbox:safe_goal/1`.

In order to run the server, a current SWI-PROLOG is required. The application is developed and tested using SWI-PROLOG 7.6.0 for Microsoft Windows. In principle, the server should also be executable with upcoming SWI-PROLOG releases. However, it should be noted, that in July 2017 a critical vulnerability was patched [Ogb17]. Previously, a Pengine was able to execute arbitrary code on the server. Therefore, the use of older versions than 7.5.12 of SWI-PROLOG is strongly discouraged. In general, the Pengines project is still under development and not yet released as a stable version. The developers explicitly point out that safety can not be guaranteed at this time [Lag17].

The server is started by running the file `'run.pl'` located in the project folder `Pengines`. In addition, a debugging mode is available (`'debug.pl'`) which offers a graphical overview of running threads on the server. Already pre-configured is the ability to run the server as a Unix daemon (see `'daemon.pl'`).

7.1.2 Client

When the server is running, the web application can be accessed locally in a web browser at `http://localhost:3030`. Recommended is a recent version of Google Chrome[15] (developed and tested using version 62.0). Of course, this is not to be understood as a limitation. The application and its functionality has also been tested in other web browsers with different rendering engines. All the latest versions of Internet Explorer 11, Microsoft Edge, and Mozilla Firefox have been successfully reviewed to check the operability of the application. According to the current usage share of desktop browsers, over 90% of web browsers used are thus covered [Net17].

The application is based on common web technologies: JavaScript, HTML5 and CSS3. In addition, several frameworks are utilised in the implementation. The layout of the application is based on MUI[16]. This is a lightweight CSS framework based on Google's Material Design Guidelines[17]. Furthermore, CodeMirror[18] is used as a editor for DCGs.

After the user has entered a DCG and formulated a request, it is executed by means of Pengines in SWI-PROLOG. The dict generated by *DCG Tracer* is returned

[15]Google Chrome: `https://www.google.de/chrome/`.
[16]MUI: `https://www.muicss.com/`.
[17]Material Design: `https://material.io/`.
[18]CodeMirror: `http://codemirror.net/`.

in JSON format. This represents the model of all required execution steps to find a solution. To render this, it is translated into HTML markup. This is achieved with the help of Handlebars[19], a logicless templating engine. Interactive features, such as displaying individual execution steps, were realised with the help of jQuery[20]. An overview of the interface and its provided functionality is given in the next section.

7.2 Overview of the Application

The application realises concepts and ideas introduced in Chapter 3. The following is an overview of the user interface and its features. In addition to this section, Appendix A.1 provides an overview of here presented features in the form of screenshots.

7.2.1 Posing Queries

The user has the opportunity to formulate a DCG himself. Alternatively, it is possible to load examples and adapt them if desired. Among others, DCGs used within this thesis for illustration purposes are available as example DCGs there.

All language components that can be processed by the presented *DCG Tracer* may be used: Terminals, non-terminals and embedded Prolog code. Terminals can be specified either as lists or as strings, the latter enclosed in double quotes. As usual in Prolog, strings are internally converted to lists of individual characters. By default, this list consists of character codes. In order to achieve a readable output, the corresponding character is displayed instead of its code.

The possibilities to make a request are based on the built-in predicate `phrase/3`. So a DCG body, an input and a rest list is needed. The DCG body is not limited to single non-terminals and may consist of a conjunction of several grammar body items. The input and rest lists can either be lists of arbitrary content, or an unbound variable. Here, specifying strings is also permitted if terminals in queried rules are based on strings or char lists.

In addition, two buttons are available: *First Solution* and *Next Solution*. The former provides the first result of a query. The latter can be used to retrieve further solutions to the current query via backtracking.

[19]Handlebars: `http://handlebarsjs.com/`.
[20]jQuery: `https://jquery.com/`.

7.2.2 Interactive Visualisation

The result of a query is displayed according to the structure presented in Section 3.2. Above the actual visualisation is an overview of the queried DCG body, the event with which the execution finished, and the input and rest list.

Underneath is the visualisation of the execution. After fetching a result, this is initialised with the final result of the query. The topmost box of the visualisation is the queried DCG body, called goals are displayed below. Of course, the visualisation is not fixed to this step, individual steps of the execution can be viewed interactively. This is done by the slider and corresponding buttons. There are buttons for step-by-step navigation through the execution steps, as well as buttons to get directly to the last or next toplevel event. The latter buttons are useful, for example, to jump back to the step where the previous solution was found. Then the following steps for finding the next solution can be viewed. This visualises backtracking: The rejection of already found solutions to goals and the execution of alternative grammar rules can be viewed step-by-step.

The current goal is highlighted in the visualisation *(green text colour)*. If the current goal is a non-terminal, the corresponding line of code is marked, too. In addition, the state of each goal is indicated by its background colour. So it is easy to distinguish whether a goal is called *(light blue)*, it could be satisfied *(dark blue)*, it failed or no further solution could be found during backtracking *(red)*, it was aborted because the recursion limit has been reached previously *(pink)*, or it has been discarded through an alternative that has been called instead *(greyed out)*. In order to be able to show for each goal in the visualisation the corresponding grammar rule, associated code is also highlighted when these are hovered.

To adjust the scale of the visualisation, there are corresponding buttons. Also, panning the visualisation is possible by dragging with the mouse.

The labeling of each box in the visualisation depends on its content: Terminals are labeled with the associated list, similarly, PROLOG code is written inside curly brackets. By default, only the functor and additional arguments of non-terminals are displayed. There is an option in the controls section to also display input and rest lists for each non-terminal. The queried DCG body is displayed analogously.

A screenshot of DCG Visualiser is given in Figure 7.1. This shows the application after fetching a result. Further insights into provided features are given in Appendix A.1 in form of screenshots.

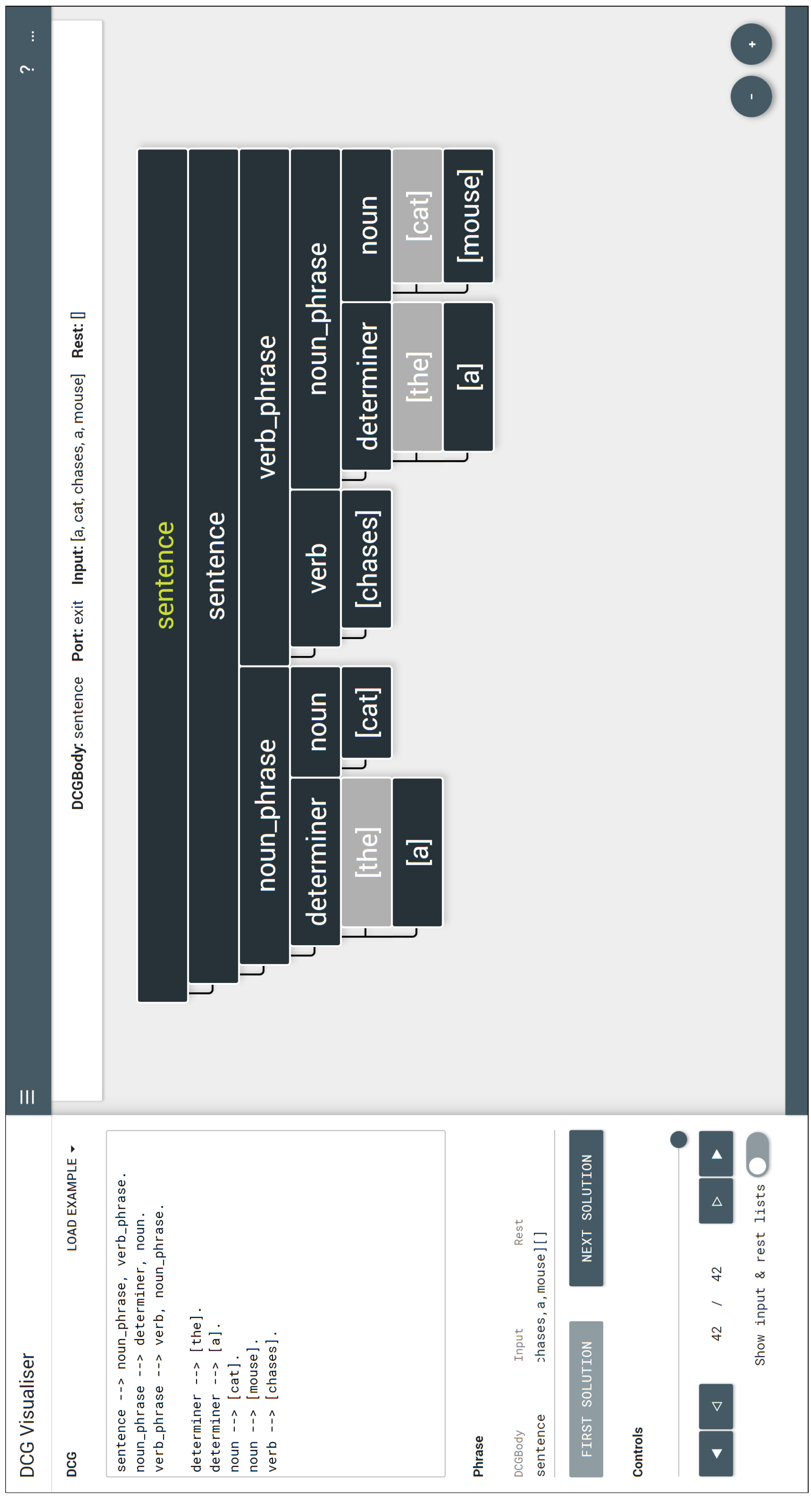


Figure 7.1: Screenshot of DCG Visualiser. Displayed is the result of a query on the DCG from Listing 4.1. The queried grammar body is `sentence//0` with the input list `[a, cat, chases, a, mouse]` and an empty rest list.

7.3 Future Work

With the developed application, it is possible to interactively comprehend the execution of queries on DCGs. Most of the features introduced in Chapter 3 have already been realised. Nevertheless, there are still ways to improve the application even further in the future.

Depending on future development of *DCG Tracer*, the presentation may change. For example, if additional control structures are implemented, they should also be visualised.

Variables in input and rest lists are unified with the corresponding result when a query is executed. Unfortunately, there is no sharing of identifiers between multiple variables in the data returned by Pengines. Thus, in each returned term, the variable naming begins with an `A` for the first variable, using the built-in `numbervars/3`. With several unbound variables, this leads to erroneous conclusions, e.g. unrelated variables appear to be unified. So the identifiers of unbound variables are not suitable to be shown. However, since the display of variable bindings and not just the output of input and rest lists is meaningful, it would be preferable to find a way how this can be realised in a proper way.

Currently, code is displayed without syntax highlighting. The library *Codemirror*, which is used for providing the text area, does not feature a language mode for highlighting PROLOG yet. There are already efforts to realise this. This is being worked on as part of SWISH.[21] The identification of individual code fragments of PROLOG code is more complex than in other programming languages. So this is currently done not locally in the browser but server-assisted. If this is released as a language mode for *CodeMirror*, it could be integrated in *DCG Visualiser*.

There is also room for improvement in the presentation of syntactical errors in the DCG and the query. Currently, errors returned by Pengines are already displayed as a warning. However, this is not further processed and thus these notifications may be confusing.

[21] SWISH: SWI-Prolog for Sharing `https://swish.swi-prolog.org/`.

8 Conclusion

This chapter gives a brief summary of this thesis. Subsequently, the future development of the presented approach and application is discussed.

8.1 Summary

In this work a visualisation of DCGs was introduced and realised. Based on existing research and the needs of the potential user group, a concept of such an application was first presented.

To realise the visualisation, different approaches for gathering data on the execution of queries on DCGs were discussed. *DCG Tracer*, a meta-interpreter based approach, was introduced and implemented. This traces the execution, subsequently a dict which represents execution steps in a structured way is created.

The thereby provided information on the query is used by the developed web application *DCG Visualiser* which displays it in an appealing manner. The connection between SWI-Prolog and JavaScript is established using Pengines. Utilising this application it is possible to interactively comprehend which execution steps Prolog performs to satisfy a query on a DCG. Backtracking and failing queries can also be visualised since failing execution steps are also displayed.

The aim of this work was to find a way of interactively visualise DCGs. This was achieved with the presented approach, resulting in an appropriate application.

8.2 Outlook

"Software is a process, it's never finished, it's always evolving. That's its nature."[22]

With the presented *DCG Tracer* and the application *DCG Visualiser* based on it, there is already a possibility of visualising DCGs. Nevertheless, there are still plenty of ideas for further improvements.

[22] Dave Winer, 1995 `http://scripting.com/davenet/1995/09/03/wemakeshittysoftware.html`.

These have already been discussed in detail in Sections 6.7 and 7.3 respectively. In summary, extending the tracer by the interpretation of further language components and subsequently integrating those into the visualisation are the next major challenges.

In addition, a study on how the presented application supports the understanding and debugging of DCGs could help to refine the direction of future development.

Currently the application is limited to the visualisation of DCGs. However, the underlying meta-interpreter based approach, as well as the representation of the execution as a dict, are flexibly expandable. Thus, based on the introduced modules and application, a visualisation of arbitrary Prolog programs could be realised in the future.

List of Figures

2.1 View of a parse tree in HDRUG 4

3.1 Entering a DCG and a query 9
3.2 Displaying a result 9
3.3 Toggle additional information 10
3.4 Showing an execution step-by-step 11
3.5 Overview of the intended application 12

4.1 The 6-port execution model 21

5.1 Example of a parse tree 33

6.1 Overview of the structure of DCG Tracer 45

7.1 Screenshot of DCG Visualiser 65

A.1 DCG Visualiser: View of a result 80
A.2 DCG Visualiser: Querying further solutions 81
A.3 DCG Visualiser: View of a failing query 82
A.4 DCG Visualiser: Detailed view of a failing step 83
A.5 DCG Visualiser: String based queries and normal PROLOG code . . . 84
A.6 DCG Visualiser: Depth limit reaching query 85
A.7 DCG Visualiser: Provided instructions 86

List of Listings

4.1 DCG for a subset of the English language 15
4.2 Modified English grammar with grammatical number 16
4.3 Syntax of DCGs 17
4.4 Declaration of `phrase/2` 21
4.5 User-defined functions on dicts 26
4.6 Using Pengines with JavaScript 28

5.1 Grammar which generates a parse tree 32
5.2 Collecting execution information using the intercepted tracer 34
5.3 Vanilla meta-interpreter suitable for DCGs 38
5.4 Meta-interpreter with tree output 39
5.5 Handling If-Then-Else statements 41
5.6 Meta-interpreter with output of failure 41

6.1 Definition of `'T'/3` 46
6.2 DCG for palindromes 47
6.3 Tracing meta-interpreter for DCGs 51

Bibliography

[Bow84] Dave Bowen. Meta-circular interpreter maintaining extended AND/OR tree. `http://www.dcs.ed.ac.uk/home/mke/edinburgh/tools/tracing/andor.pl`, 1984. Accessed: 21.10.2017.

[Bra12] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Pearson Education, 2012.

[Byr80] Lawrence Byrd. Understanding the control flow of prolog programs. *Logic Programming Workshop*, 1980.

[CM03] William Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[CR96] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996.

[DN94] Mireille Ducassé and Jacques Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19:351–384, 1994.

[Dod92] Tony Dodd. DCGs in ISO Prolog – a proposal. Standard, BSI, 1992.

[EB88] Marc Eisenstadt and Mike Brayshaw. The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *The Journal of Logic Programming*, 5(4):277–342, 1988.

[Fla94] Peter A Flach. *Simply Logical – intelligent reasoning by example*. John Wiley & Sons, 1994.

[Gav17] Marco Gavanelli. SLDNF-Draw: Visualization of Prolog operational semantics in LaTeX. *Intelligenza Artificiale*, 11(1):81–92, 2017.

[Hod15] Jonathan Hodgson. ISO/IEC DTR 13211-3:2006 Definite clause grammar rules. Standard, International Organization for Standardization, 2015.

[Imp14] Implementing cut in tracing meta interpreter prolog. https://stackoverflow.com/questions/27235148/implementing-cut-in-tracing-meta-interpreter-prolog, 2014. Accessed: 24.10.2017.

[ISO95] ISO. Information technology – Programming languages – Prolog – Part 1: General core. Standard ISO/IEC 13211-1:1995, International Organization for Standardization, 1995.

[Kah84] Kenneth Kahn. A grammar kit in prolog. In *New Horizons in Educational computing*. Halsted Press, 1984.

[KCT16] Stefan Kögel, Joscha Cüppers, and Matthias Tichy. ClickyEvaluation: A Step-by-Step Evaluator for Functional Programming Expressions. In *2nd European Conference of Software Engineering Education*. ECSEE, 2016.

[KK71] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. In *Artificial Intelligence 2*, pages 227–260. North-Holland Publishing Company, 1971.

[KNN99] Gabriella Kókai, Jörg Nilson, and Christian Niss. GIDTS: A graphical programming environment for prolog. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 95–104. ACM, 1999.

[KSS$^+$] Tom Klonikowski, Jan Segre, Dmitry Soloviev, Johny Jose, and Raivo Laanemets. node-swipl: A node.js interface to the SWI-Prolog library. https://github.com/rla/node-swipl. Accessed: 22.10.2017.

[Laa] Raivo Laanemets. node-swipl-stdio: A Node.js interface to the SWI-Prolog communicating over stdio. https://github.com/rla/node-swipl-stdio. Accessed: 22.10.2017.

[Lag17] Torbjörn Lager. Getting started with pengines. http://pengines.swi-prolog.org/docs/getting_started.html, 2017. Accessed: 08.11.2017.

[LF11] Adam Lally and Paul Fodor. Natural language processing with prolog in the IBM watson system. *The Association for Logic Programming (ALP) Newsletter*, 2011.

[LW14] Torbjörn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. *TPLP*, 14(4-5):539–552, 2014.

[LW15] Zhixuan Lai and Alessandro Warth. Prolog Visualizer. http://cdglabs.org/prolog/, 2015. Accessed: 29.09.2017.

[MD99] Sarah Mallet and Mireille Ducassé. Myrtle: A set-oriented meta-interpreter driven by a "relational" trace for deductive databases debugging. In *Research Report RR-3598*. INRIA, 1999.

[MD10] Paulo Moura and Klaus Daessler. ISO/IEC DTR 13211-3:2006 Definite clause grammar rules. Standard, International Organization for Standardization, 2010.

[MSL10] Lee Mondshein, Abdul Sattar, and Torben Lorenzen. Visualizing prolog: a jigsaw puzzle approach. *ACM Inroads*, 1(4):43–48, 2010.

[Net17] NetMarketShare. Desktop Top Browser Share Trend. `https://netmarketshare.com/`, 2017. Accessed: 31.10.2017.

[Neu15] Ulrich Neumerkel. SWI7 and ISO Prolog. `https://www.complang.tuwien.ac.at/ulrich/iso-prolog/SWI7_and_ISO`, 2015. Accessed: 30.10.2017.

[NKS17] Falco Nogatz, Jona Kalkus, and Dietmar Seipel. Declarative XML Schema Validation with SWI-Prolog. In *Proceedings of 25th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2017)*, 2017.

[Ogb17] Anne Ogborn. Security vulnerability in pengines. `http://www.swi-prolog.org/news/4bdf6790-6d90-11e7-aede-00163e986a2a`, July 2017. Accessed: 08.11.2017.

[Ohma] Ohm Editor. `https://ohmlang.github.io/editor/`. Accessed: 29.09.2017.

[Ohmb] Ohm Visualizer. `http://www.cdglabs.org/ohm/visualizer/`. Accessed: 29.09.2017.

[PB87] Helen Pain and Alan Bundy. What stories should we tell novice prolog programmers. *Artificial intelligence programming environments*, pages 119–130, 1987.

[PW80] Fernando CN Pereira and David HD Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3):231–278, 1980.

[SAF17] Nada Sharaf, Slim Abdennadher, and Thom Frühwirth. Using rules to animate prolog programs. *CEUR*, 2017.

[Sei02] Dietmar Seipel. Processing XML-Documents in Prolog. In *Proceedings of 17th Workshop on Logic Programming (WLP 2002)*, 2002.

[Shi89] Hideaki Shinomi. Graphical representation and execution animation for prolog programs. In *International Workshop on Industrial Applications of Machine Intelligence and Vision*, pages 181–186. IEEE, 1989.

[SMM09] Alessia Stalla, Viviana Mascardi, and Maurizio Martelli. PrettyProlog: A Java interpreter and visualizer of Prolog programs. *Atti del convegno CILC*, 9:7–11, 2009.

[SS86] Leon Sterling and Ehud Y Shapiro. *Prolog: fortgeschrittene Programmiertechniken*. Addison-Wesley, 1986.

[SS94] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[TAK95] Dan E Tamir, Ravi Ananthakrishnan, and Abraham Kandel. A visual debugger for pure prolog. *Information Sciences-Applications*, 3(2):127–147, 1995.

[Tay88] Josephine Annette Taylor. *Programming in Prolog: an in-depth study of problems for beginners learning to program in Prolog*. PhD thesis, University of Sussex, 1988.

[TdBP99] Chris Taylor, Benedict du Boulay, and Mukesh J Patel. A revised textual tree trace notation for prolog. In *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, pages 267–281. Ablex Publishing Corporation, 1999.

[Tri17] Markus Triska. The Power of Prolog. `https://www.metalevel.at/prolog`, 2005–2017. Accessed: 07.10.2017.

[VN] Gertjan Van Noord. Hdrug example. `http://odur.let.rug.nl/~vannoord/Hdrug/example.html`. Accessed: 21.10.2017.

[VNB97] Gertjan Van Noord and Gosse Bouma. Hdrug. A Flexible and Extendible Development Environment for Natural Language Processing. *Computational Environments for Grammar Development and Linguistic Engineering*, 1997.

[WHVDM08] Jan Wielemaker, Zhisheng Huang, and Lourens Van Der Meij. Swi-prolog and the web. *Theory and practice of logic programming*, 8(3):363–392, 2008.

[Wie] Jan Wielemaker. Swi-prolog future directions. `http://www.swi-prolog.org/Directions.html`. Accessed: 30.10.2017.

[Wie14] Jan Wielemaker. Swi-prolog version 7 extensions. In *Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014*, pages 109–123, 2014.

[Wie17a] Jan Wielemaker. Swi prolog: 7.6.0-rc1 available. `http://www.swi-prolog.org/news/10c66b98-98a3-11e7-bab4-00163e986a2a`, September 2017. Accessed: 08.11.2017.

[Wie17b] Jan Wielemaker. *SWI Prolog Reference Manual, Updated for version 7.6, October 2017*, 2017.

[Wpc] Alessandro Warth and Ohm project contributors. harc/ohm: A library and language for building parsers, interpreters, compilers, etc. `https://github.com/harc/ohm`. Accessed: 29.09.2017.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

Appendix

A.1 Screenshots of DCG Visualiser

The following pages provide an overview of the application *DCG Visualiser*. For this purpose, different features are presented by means of screenshots. Utilised DCGs are mostly already introduced within this thesis. In addition, these are provided in the application as example DCGs.

- **View of a result:**
  Figure A.1, Page 80
- **Querying further solutions:**
  Figure A.2, Page 81
- **View of a failing query:**
  Figure A.3, Page 82
- **Detailed view of a failing step:**
  Figure A.4, Page 83
- **String based queries and normal Prolog code:**
  Figure A.5, Page 84
- **Depth limit reaching query:**
  Figure A.6, Page 85
- **Provided instructions:**
  Figure A.7, Page 86

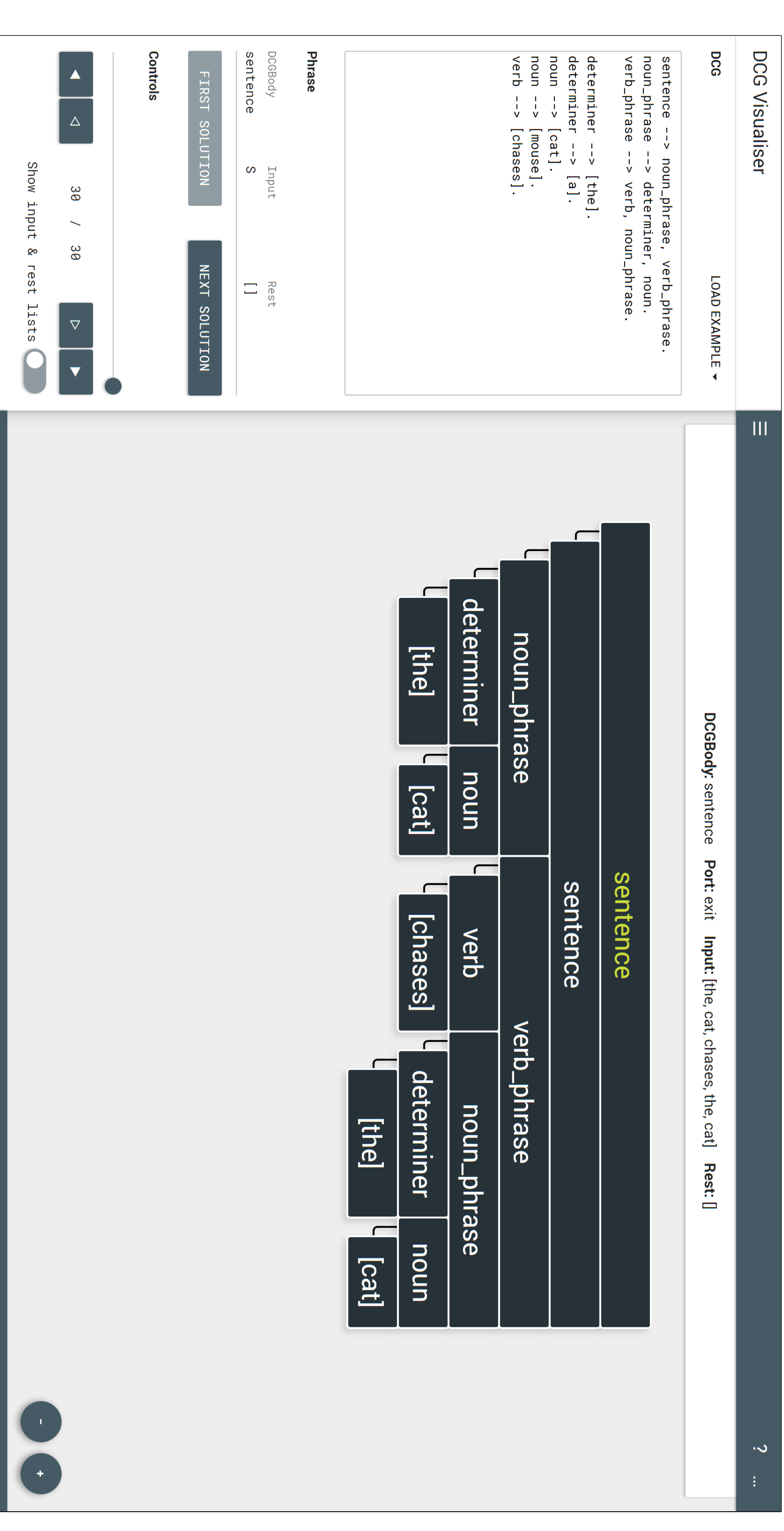


Figure A.1: This picture shows the first result of a query on the DCG presented in Listing 4.1. The grammar body `sentence//0` is queried, visualised below it are the call of the corresponding non-terminal and resulting subgoals to find a solution.

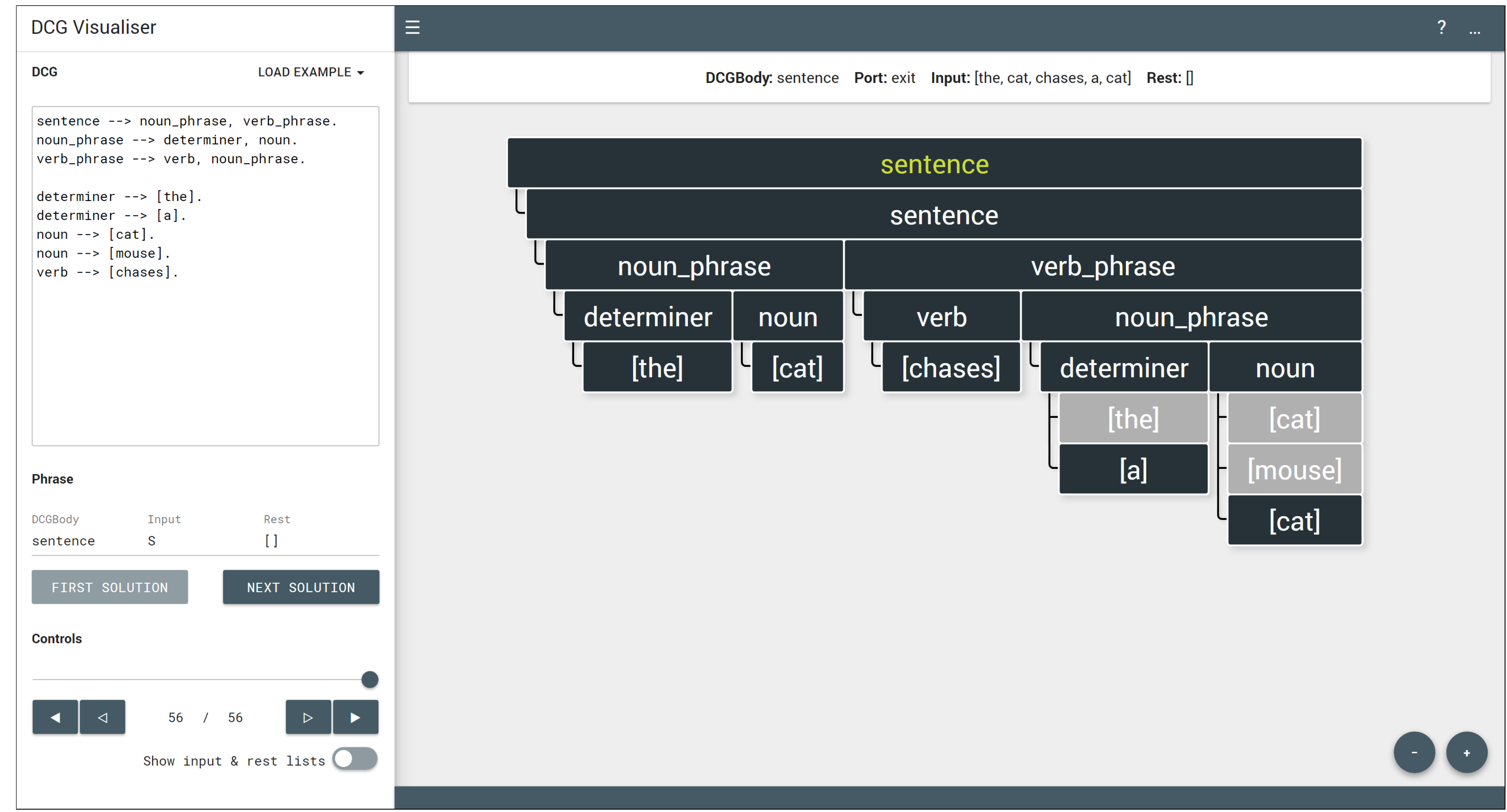


Figure A.2: After a query succeeded, it is possible to search for further solutions via backtracking. The output of the third solution to the request from Figure A.1 is shown in this figure.

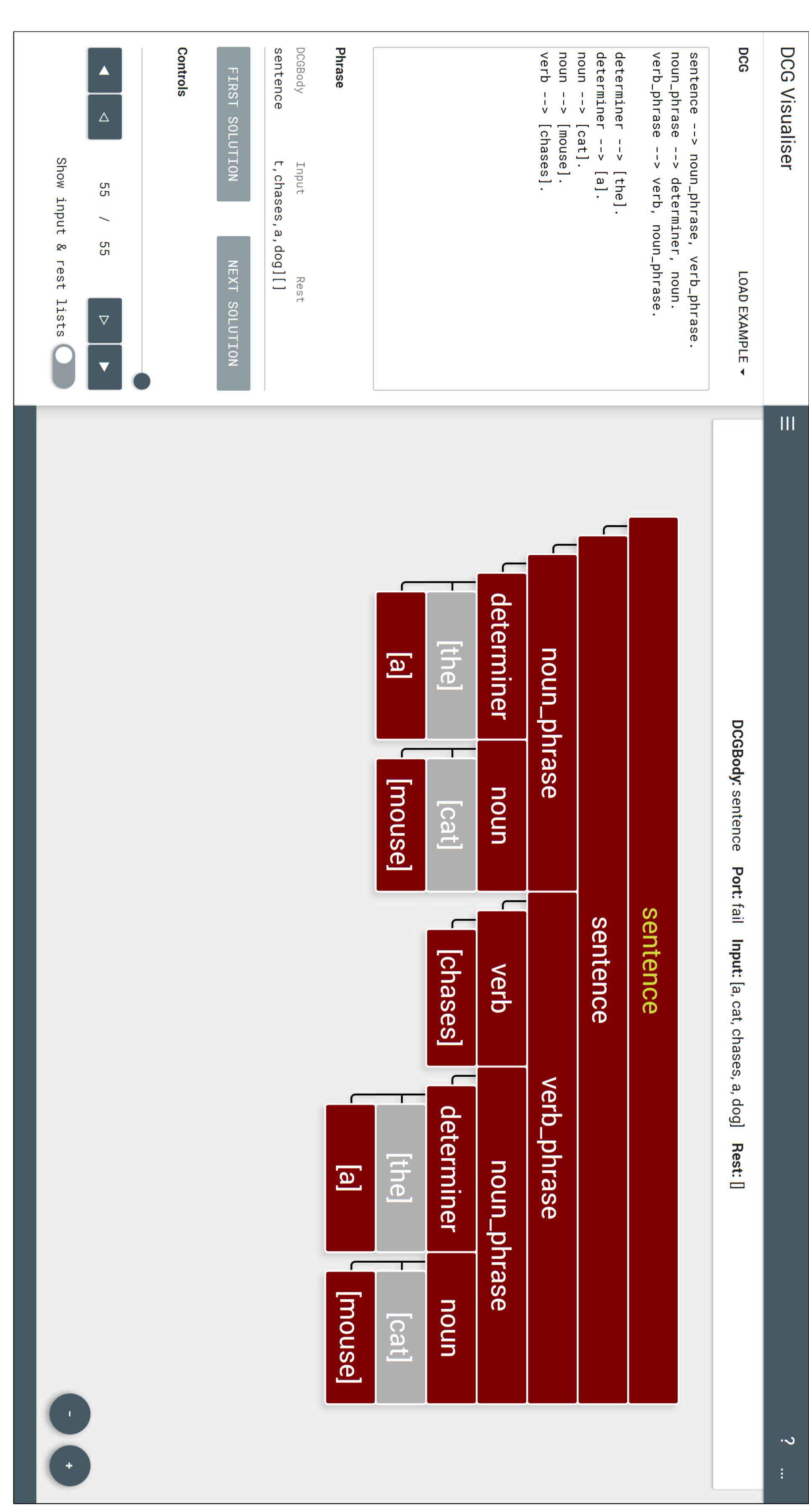


Figure A.3: In the case of failed queries, the final result is shown first also: All called goals have failed on backtracking, because no succeeding alternative could be found for any of the invoked goals.

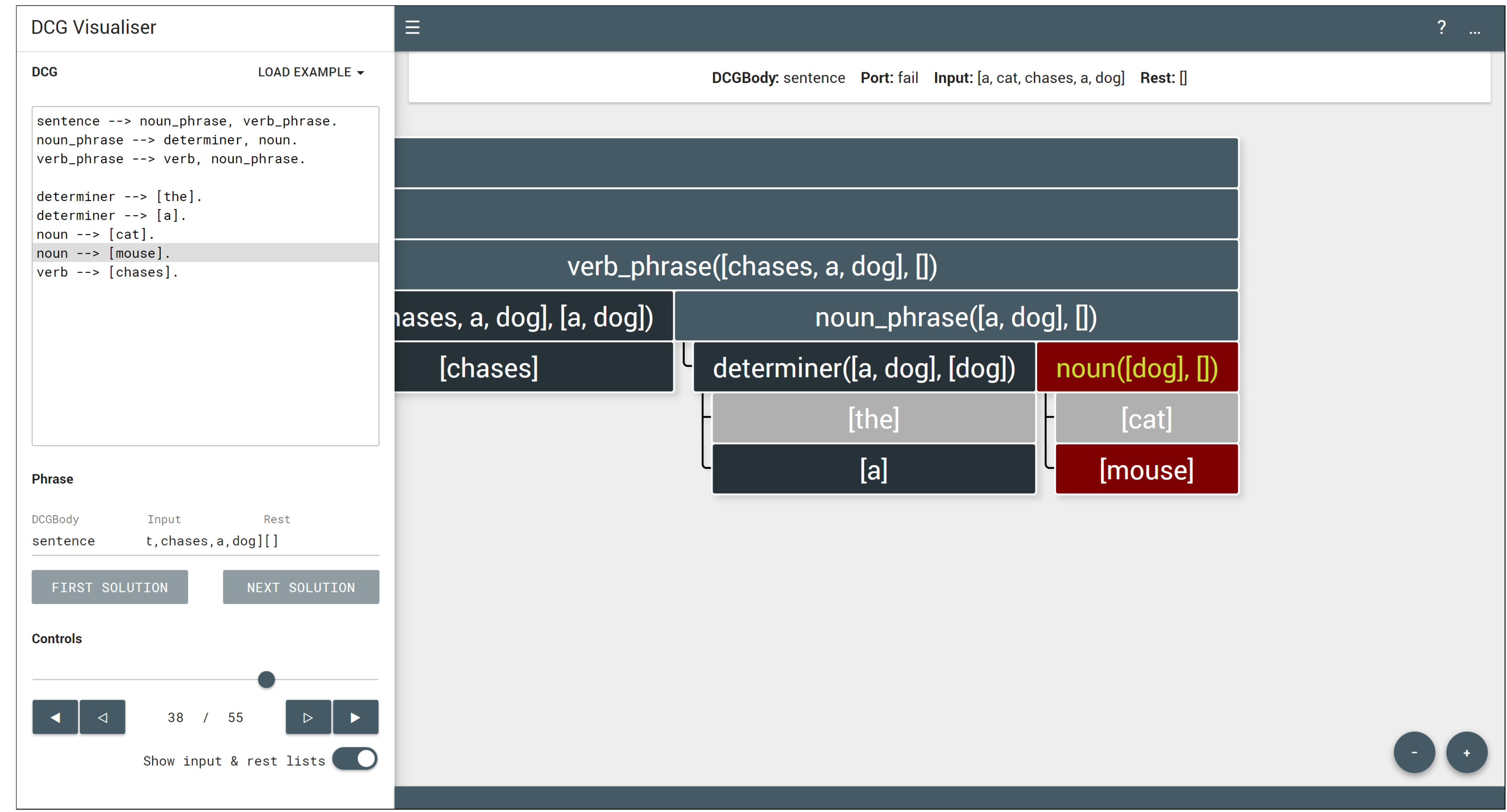


Figure A.4: The visualisation allows the inspection of a failed execution step: Since intermediate steps can be viewed, the origin of failure in Figure A.3 and its associated rule can be identified. In addition, the input and rest lists are displayed here.

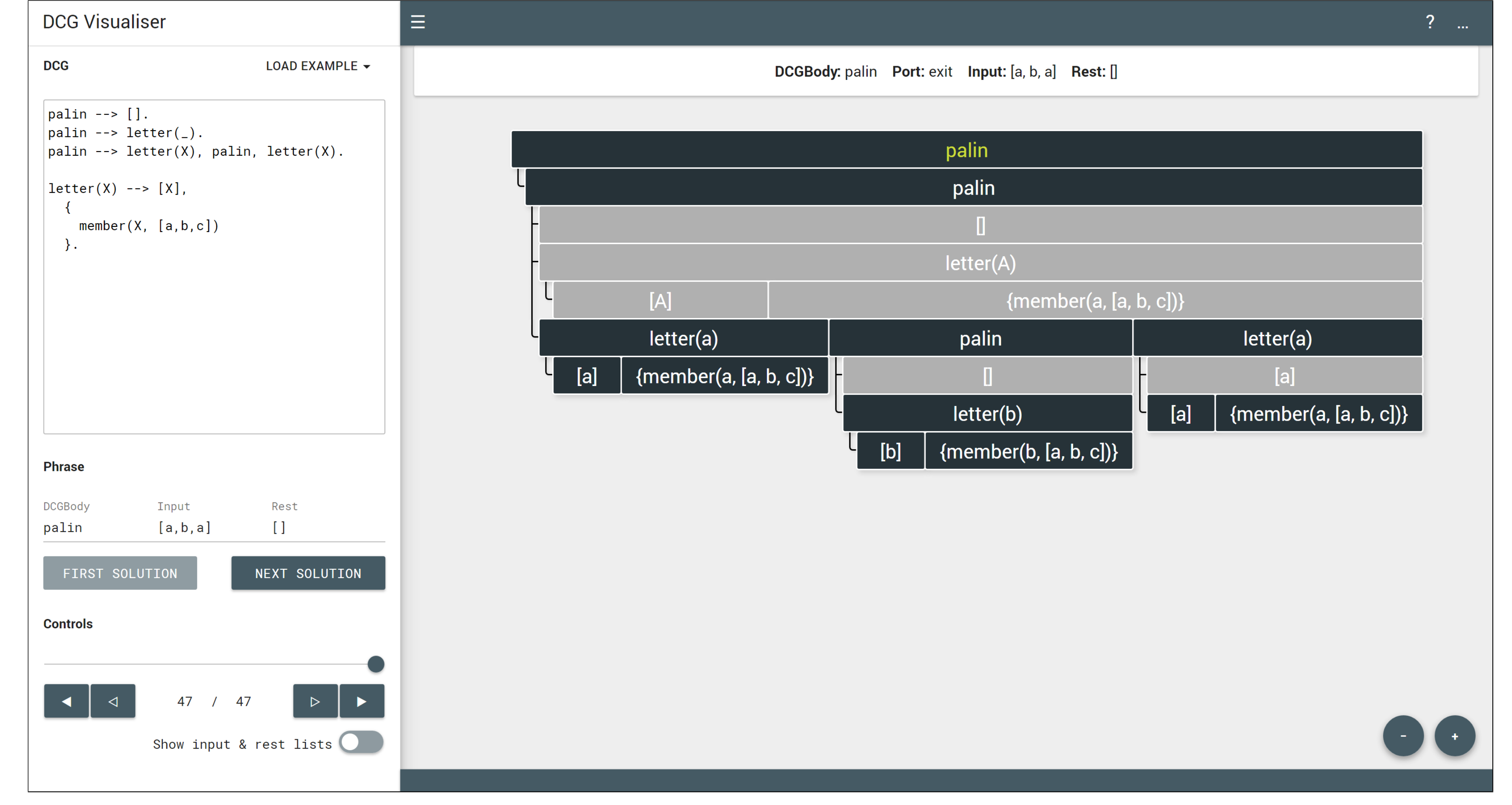


Figure A.5: Visualisation of the result of a query which uses a string as input list. The queried DCG describes palindromes and makes use of embedded PROLOG code, which is evaluated and its result displayed.

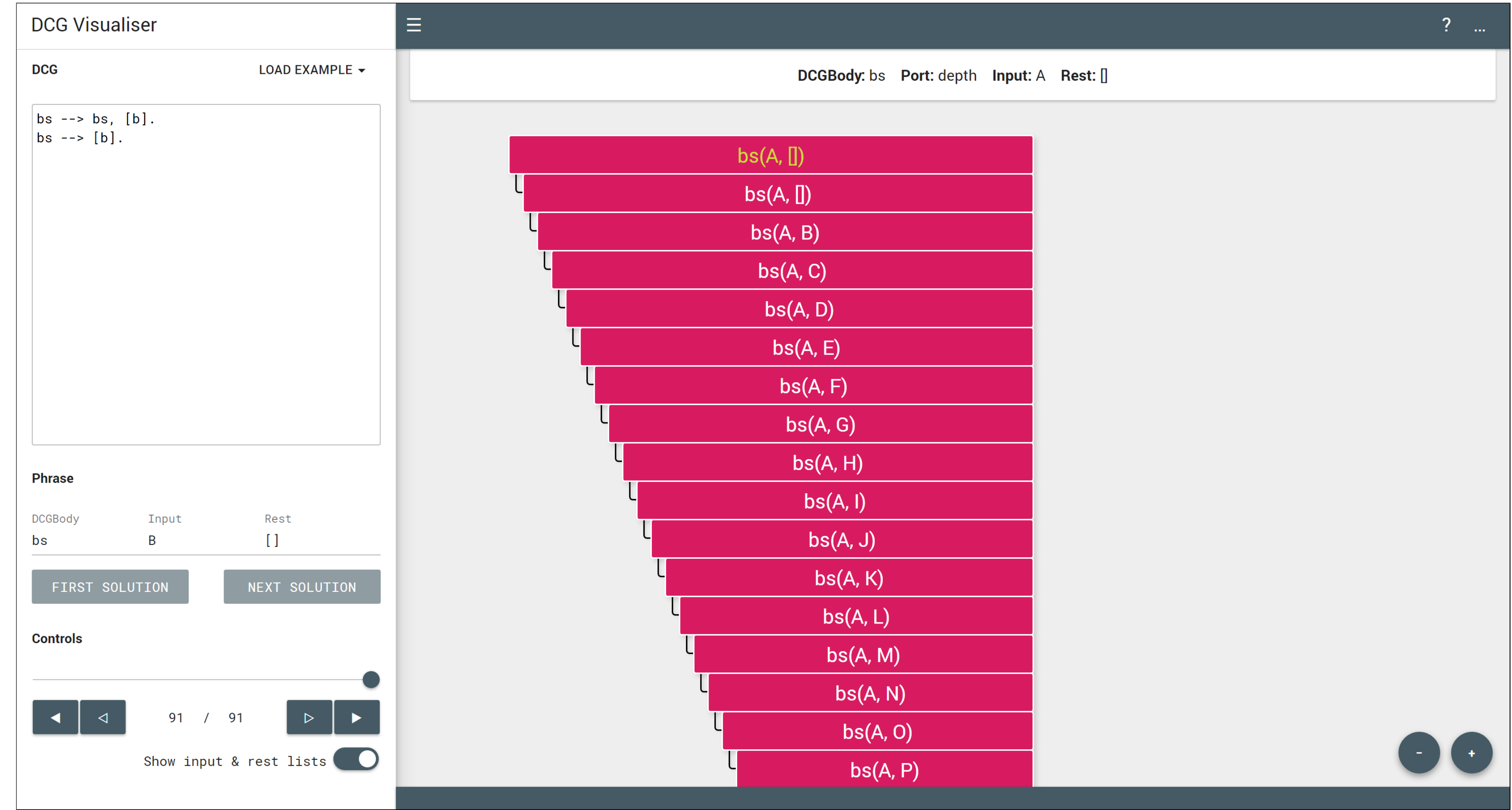


Figure A.6: This screenshot shows the call of a left-recursive grammar. The specified depth limit cancels queries such as this. Thereby cancelled goals are marked accordingly in pink colour.

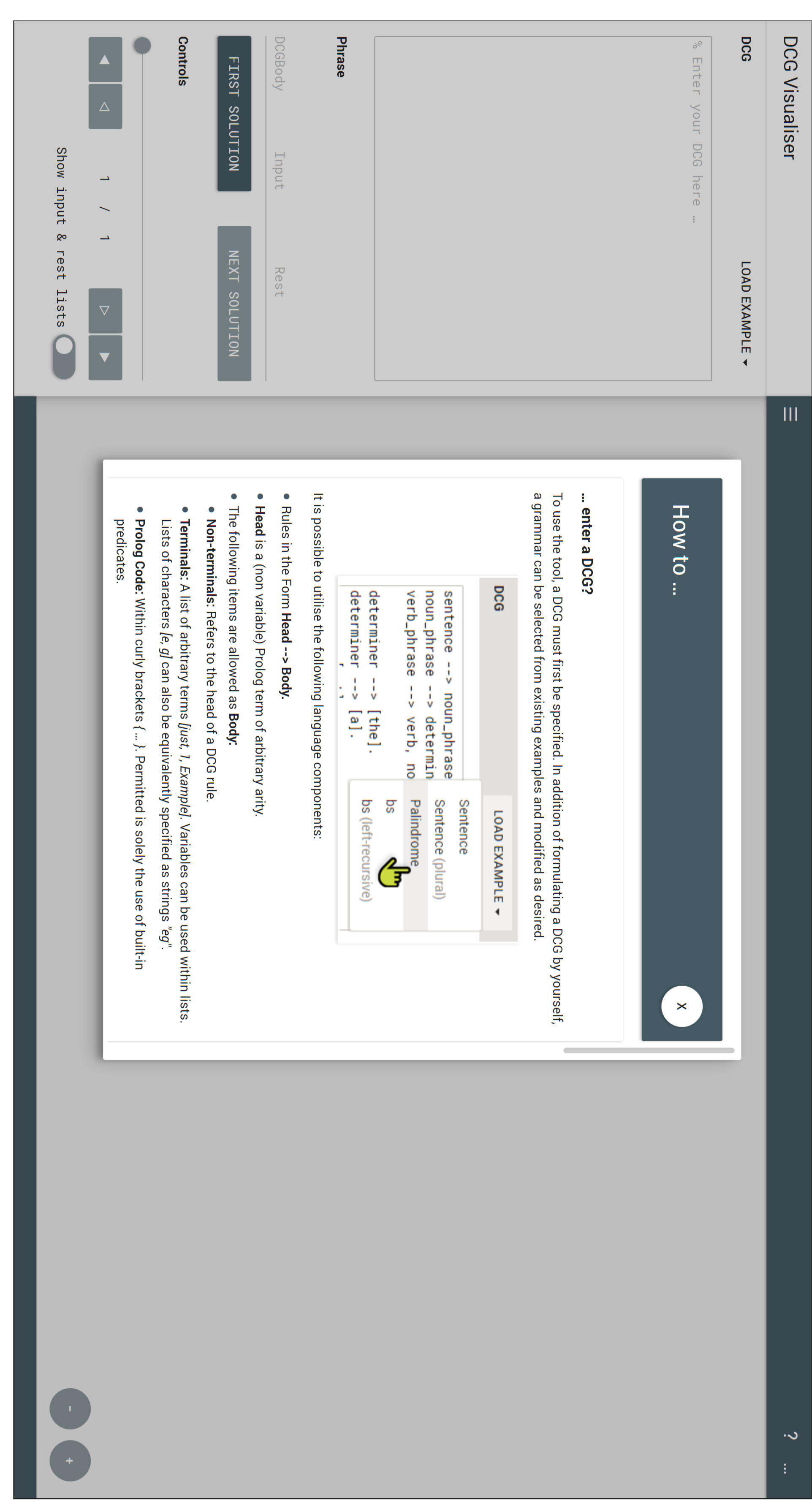


Figure A.7: An overview about how the application can be used and how results are visualised is given in the provided help.

A.2 CD Contents

The provided CD contains the following data:

- **/dcg_tracer:**
  Prolog modules for analysing DCG execution
  (as introduced in Section 6.6).
- **/dcg_visualiser:**
  A Pengines server and the web application DCG Visualiser
  (as introduced in Chapter 7).
- **/listings:**
  Example DCGs and code which is used within the thesis,
  but not in the actual implementation.
- **/screenshots:**
  Screenshots of the visualisation.
  These images are also included in Appendix A.1 of this thesis.
- **thesis.pdf:**
  A digital copy of this thesis.

For more information about folder contents, see the README files stored in each of the folders on the CD.

Erklärung

Ich, Jona Kalkus, Matrikel-Nr. 1839485, versichere hiermit, dass ich meine Masterarbeit mit dem Thema

An Interactive Visualisation for Definite Clause Grammars

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Masterarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Universität abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Würzburg, den 9.11.2017

Jona Kalkus