Defining and Implementing Domain-Specific Languages with Prolog

— Dissertation —

Falco Nogatz

January 2022

Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik

Supervisor: Prof. Dr. Dietmar Seipel

To my loving wife Tessa.

Abstract

The landscape of today's programming languages is manifold. With the diversity of applications, the difficulty of adequately addressing and specifying the used programs increases. This often leads to newly designed and implemented domain-specific languages. They enable domain experts to express knowledge in their preferred format, resulting in more readable and concise programs. Due to its flexible and declarative syntax without reserved keywords, the logic programming language Prolog is particularly suitable for defining and embedding domain-specific languages.

This thesis addresses the questions and challenges that arise when integrating domain-specific languages into Prolog. We compare the two approaches to define them either externally or internally, and provide assisting tools for each. The grammar of a formal language is usually defined in the extended Backus–Naur form. In this work, we handle this formalism as a domain-specific language in Prolog, and define term expansions that allow to translate it into equivalent definite clause grammars. We present the package *library(dcg4pt)* for SWI-Prolog, which enriches them by an additional argument to automatically process the term's corresponding parse tree. To simplify the work with definite clause grammars, we visualise their application by a web-based tracer.

The external integration of domain-specific languages requires the programmer to keep the grammar, parser, and interpreter in sync. In many cases, domain-specific languages can instead be directly embedded into Prolog by providing appropriate operator definitions. In addition, we propose syntactic extensions for Prolog to expand its expressiveness, for instance to state logic formulas with their connectives verbatim. This allows to use all tools that were originally written for Prolog, for instance code linters and editors with syntax highlighting. We present the package *library(plammar)*, a standard-compliant parser for Prolog source code, written in Prolog. It is able to automatically infer from example sentences the required operator definitions with their classes and precedences as well as the required Prolog language extensions. As a result, we can automatically answer the question: Is it *possible* to model these example sentences as valid Prolog clauses, and *how*?

We discuss and apply the two approaches to internal and external integrations for several domain-specific languages, namely the extended Backus–Naur form, GraphQL, XPath, and a controlled natural language to represent expert rules in if-then form. The created toolchain with *library(dcg4pt)* and *library(plammar)* yields new application opportunities for static Prolog source code analysis, which we also present.

Zusammenfassung

Die Landschaft der heutigen Programmiersprachen ist vielfältig. Mit ihren unterschiedlichen Anwendungsbereichen steigt zugleich die Schwierigkeit, die eingesetzten Programme adäquat anzusprechen und zu spezifizieren. Immer häufiger werden hierfür domänenspezifische Sprachen entworfen und implementiert. Sie ermöglichen Domänenexperten, Wissen in ihrem bevorzugten Format auszudrücken, was zu lesbareren Programmen führt. Durch ihre flexible und deklarative Syntax ohne vorbelegte Schlüsselwörter ist die logische Programmsprache Prolog besonders geeignet, um domänenspezifische Sprachen zu definieren und einzubetten.

Diese Arbeit befasst sich mit den Fragen und Herausforderungen, die sich bei der Integration von domänenspezifischen Sprachen in Prolog ergeben. Wir vergleichen die zwei Ansätze, sie entweder extern oder intern zu definieren, und stellen jeweils Hilfsmittel zur Verfügung. Die Grammatik einer formalen Sprache wird häufig in der erweiterten Backus–Naur–Form definiert. Diesen Formalismus behandeln wir in dieser Arbeit als eine domänenspezifische Sprache in Prolog und definieren Termexpansionen, die es erlauben, ihn in äquivalente *Definite Clause Grammars* für Prolog zu übersetzen. Durch das Modul *library(dcg4pt)* werden sie um ein zusätzliches Argument erweitert, das den Syntaxbaum eines Terms automatisch erzeugt. Um die Arbeit mit Definite Clause Grammars zu erleichtern, visualisieren wir ihre Anwendung in einem webbasierten Tracer.

Meist können domänenspezifische Sprachen jedoch auch mittels passender Operatordefinitionen direkt in Prolog eingebettet werden. Dies ermöglicht die Verwendung aller Werkzeuge, die ursprünglich für Prolog geschrieben wurden, z.B. zum Code-Linting und Syntax-Highlighting. In dieser Arbeit stellen wir den standardkonformen Prolog-Parser *library(plammar)* vor. Er ist in Prolog geschrieben und in der Lage, aus Beispielsätzen automatisch die erforderlichen Operatoren mit ihren Klassen und Präzedenzen abzuleiten. Um die Ausdruckskraft von Prolog noch zu erweitern, schlagen wir Ergänzungen zum ISO Standard vor. Sie erlauben es, weitere Sprachen direkt einzubinden, und werden ebenfalls von *library(plammar)* identifiziert. So ist es bspw. möglich, logische Formeln direkt mit den bekannten Symbolen für Konjunktion, Disjunktion, usw. als Prolog-Programme anzugeben.

Beide Ansätze der internen und externen Integration werden für mehrere domänenspezifische Sprachen diskutiert und beispielhaft für GraphQL, XPath, die erweiterte Backus–Naur–Form sowie Expertenregeln in Wenn–Dann–Form umgesetzt. Die vorgestellten Werkzeuge um *library(dcg4pt)* und *library(plammar)* ergeben zudem neue Anwendungsmöglichkeiten auch für die statische Quellcodeanalyse von Prolog-Programmen.

Acknowledgements

Writing this thesis would not have been possible without the constant help and support granted to me by numerous people. First of all, I would like to thank my supervisor, Dietmar Seipel, for six years of guidance and support. I am grateful for the sharing of ideas, your patience, as well as your confidence in my progress as a researcher.

I would also like to thank my fantastic co-authors Salvador Abreu, Thom Frühwirth, Sebastian Krings, and Philipp Körner for their help in writing and collaborating on a broad variety of papers. All the articles incorporated in this thesis have benefited from suggestions and constructive criticism given by the anonymous reviewers of WLP 2016, SLATE 2016, LTC 2017, WLP 2017, RuleML+RR 2018, PPDP 2018, SLATE 2019, and ICLP 2019 as well as the COMLAN journal, for whose time and work I am grateful.

Furthermore, I would like to express my gratitude to my family and friends who withstood all the annoyances my commitment to this work caused at times. Thanks to my friends and former colleagues from the University of Ulm, who inspire me still today. Finally, I am deeply indebted to my wife, Tessa, for your love, your care, and your support that has helped throughout the entire period.

During my time in Würzburg, I was lucky enough to participate actively in the institute's teaching activities, and I would like to thank all the students I taught and supervised for the interesting times. In particular, I would like to express my gratitude towards Julia Kübert, Jona Kalkus, Lucas Kinne, Kevin Jonscher, and Daniel Haumann. Their work contributed to the ideas presented in this thesis as well as to joint publications and open-source software.

The work at hand has been proofread by Petra Gospodnetic, Timo Oess, and Tessa Nogatz. Your reviews on parts of this thesis helped me fix many errors. Thank you!

Finally, I would like to express my gratitude to the assessment committee for your time and your invaluable comments and recommendations.

Contents

Nomenclature

General

EBNF	Extended Backus–Naur Form
AOT	Ahead-of-Time Compiler
API	Application Programming Interface
AST	Abstract Syntax Tree
CNL	Controlled Natural Language
CSP	Constraint Satisfaction Problem
CST	Concrete Syntax Tree
DSL	Domain-Specific Language
GPL	General-Purpose Language
JIT	Just-in-Time Compiler
MI	Meta-Interpreter
REPL	Read–Eval–Print Loop
W3C	World Wide Web Consortium

Computer and Domain-Specific Languages

JSON	JavaScript Object Notation
ACE	Attempto Controlled English
CHR	Constraint Handling Rules
CSV	Comma-Separated Values
DCG	Definite Clause Grammar
DTD	Document Type Definition
MathML	Mathematical Markup Language
PEG	Parsing Expression Grammar
PHP	PHP: Hypertext Preprocessor *(recursive acronym)*
RDF	Resource Description Format
SGML	Standard Generalized Markup Language
TAP	Test Anything Protocol
XML	Extensible Markup Language
XSD	XML Schema Definition

1

Introduction

What gets created depends on who is creating it.

— Verónica Dahl[1]

In a world of about 200 countries, there are more than 6000 spoken languages today. But this number is in a high state of flux: at the end of the 21st century, half of the spoken languages in the world could be extinct by the current trend [46]. Every language that is not a language of government, commerce, or wider communication, is threatened in the modern world. Though the omnipresence of world-wide communications using the internet lowers the bar for distributed work teams across countries, it requires common languages. This again results in a downward spiral for language minorities.

While the number of spoken languages decreases, the opposite is true for programming languages, as can be seen in the usage statistics of GitHub (https://github.com/). GitHub is the largest Git repository hosting service in the world with more than 96 million projects, with contributions by around 40 million developers. In 2017, the open-source projects hosted at GitHub were reported to be written in 337 unique programming languages. In the recent four years, this number increased by more than 10 %.[2] All in all, the English Wikipedia lists more than 700 programming languages of historic and current use,[3] which does not even include markup languages like HTML or LaTeX, and data formats and their applications, e. g., XML and MathML.

[1] Quote from "Founding Mother of Logic Programming", https://medium.com/a-computer-of-ones-own/verónica-dahl-founding-mother-of-logic-programming-c8ccaee969bb. Verónica Dahl is an Argentine/Canadian computer scientist and was one of the first female graduates at the Université d'Aix-Marseille in France to receive a doctorate in Artificial Intelligence. In 1997, she was honoured by the Association for Logic Programming (ALP) as one of 15 scientists recognised as *Pioneers of Logic Programming*.

[2] GitHub publishes annual reports called *GitHub Octoverse* with statistics about the usage of their services and the hosted projects. The latest report, covering the data of the year 2021, is available online at https://octoverse.github.com/. Older versions can be accessed as https://octoverse.github.com/2020/ and similar.

[3] List of programming languages, https://en.wikipedia.org/wiki/List_of_programming_languages

The increasing number of *computer languages* – to subsume programming and markup languages – has multiple reasons. Firstly, the creation of a new language has become easy. With the help of parser generators like ANTLR,[4] it just needs the definition of the language's syntax as a context-free grammar to generate an appropriate lexer and parser. But even without a deeper understanding of grammars and formal languages, new languages and data formats can be easily created by simply restricting existing languages to fit in a particular application domain. For instance, the *mathematical markup language* MathML [54] is an application of XML for describing mathematical notations. It is a strict subset of the popular *extensible markup language* XML [11], hence respecting all of XML's syntax requirements. MathML only further restricts the document's structure to a well-defined set of allowed attributes, elements, and data types. These additional requirements are specified in a Relax NG Schema [119], which is similar to the more popular XML Schema [35] and *document type definition* (DTD).

A second reason for the growing number of computer languages is the rise of computers and applications in all areas of our lives, resulting in an increasing variety of application domains and requirements. The latter encourages the creation of new programming and markup languages. Like the aforementioned MathML, a language might be tailored to a specific application domain. So-called *domain-specific languages* (DSLs) have been used in various areas such as graphics, financial, high-performance computing, and many others. Because of their well-defined purposes, these domain-specific languages often increase productivity, reliability, maintainability, and flexibility in comparison to *general-purpose languages*. A literature overview of the topic of domain-specific languages is given in [120]. It highlights the advantages of working with different languages in a single software project, as well as the usefulness of defining new languages only for a particular field of applications, and possible means for their integration into existing software systems.

In this thesis, we focus on the definition, implementation, and usage of domain-specific languages in Prolog. This chapter gives an introduction to the work at hand. Section 1.1 emphasises the use of Prolog to define and use domain-specific languages. In Section 1.2, we formulate the goals of this thesis as well as the addressed problems. The main results are summarised in Section 1.3. In Section 1.4, we give an overview of the subsequent chapters and their contents. It also introduces important notions, terms and abbreviations that are used throughout the work. Section 1.5 makes the connection from the chapters to our supporting contributions, which have been published in journals and conference proceedings, as well as the created software that has been published under open-source licenses on GitHub.

[4] ANTLR (*Another Tool for Language Recognition*, [93]), https://www.antlr.org/, is a widely-used Java-based parser generator.

1.1. Motivation

Although the toolset to create a domain-specific language has been improved, the overall process is still a challenging software engineering task – beginning with the definition of the requirements, to the language's implementation, and finally its integration into an existing software environment, which always requires *glue code* to a general-purpose language. Prolog has been proven useful as an integrating tool, supporting a wide range of external resources [140]. Not only because of Prolog's long history as a programming language, but also because systems like SWI-Prolog provide libraries and built-in predicates to access different data stores and data formats.

In addition, Prolog provides several means to parse and process other computer languages. All major Prolog systems ship with support for *definite clause grammars* (DCG) [57, 96]. DCGs have been subject of research since 1980. They are a powerful mean to describe grammars in first-order logic, and bring all of Prolog's built-in capabilities to drive the parsing process. With its support for logic variables, unification, and backtracking, DCGs allow to define powerful parsers that are not restricted to context-free grammars.

Defined in this way, the code of a domain-specific language can be put in a separate file, and read in and parsed by Prolog. This is the traditional way to integrate *external* languages into Prolog. However, with *quasi-quotations*, which were introduced in SWI-Prolog in 2013 [137], domain-specific languages can also be directly embedded into normal Prolog code, which lowers the barrier for both the creators of the domain-specific language, as well as for Prolog developers integrating it with the host language. Although quasi-quotations are not yet part of the ISO Prolog standard [55], they are a syntactic enhancement particularly suited for the work with domain-specific languages, as they overcome Prolog's traditionally poor multi-line string handling and the need for otherwise escaped special characters and line-endings. As a result, code of the external domain-specific language can be put literally next to the program logic specified in the host language Prolog, without the need of further adjustments.

However, not all domain-specific languages require such a traditional approach of integration in the host language Prolog. Instead, some computer languages might already be a subset of Prolog or need only minor adjustments to be so. Given that, their sentences could be directly taken as Prolog terms, resulting in an *internal* language. This way, we can omit the challenging and often error-prone process of formally defining the DSL's syntax in the form of a context-free or a definite clause grammar, since the internal integration of a DSL does not rely on a user-defined parser.

After having defined appropriate prefix, suffix, and infix operators, the actual parsing is left to the Prolog system; the resulting term is later used by an interpreter to process the encoded information. This approach is similar to the aforementioned example of MathML, which is compatible with all tools that were originally designed to work with XML. For instance, it is not necessary to build separate code formatters for MathML, given that the ones already available for XML are suitable as well.

In contrast to imperative programming languages, Prolog comes with a *flexible syntax without keywords*, thus providing the possibility to define the domain-specific language internally. This way, Prolog is suitable as a host for integrating other languages. Although the approach to define an internal domain-specific language requires the adherence to Prolog's syntax, it relieves the programmer from the burden of keeping in sync three important parts – a user-defined grammar, its corresponding parser, and a subsequent interpreter. In addition, combined with Prolog's macro-like *term and goal expansion*, the Prolog term representing the domain-specific language can be modified at compile-time. It is not restricted to the structure defined by the operators, which are originally just necessary to allow the embedding of the domain-specific language into Prolog.

The syntax of a domain-specific language and its means for integration into an existing software environment, independent of the used host language, are important criteria in the acceptance of a newly created language or data format and should not be underestimated. The syntax should be as close as possible to the conventional notation used in the application domain, while still being easy to adapt for experts of the host language. In this regard Prolog, with its flexible syntax and well-established tools to create expressive parsers, has clear advantages over imperative programming languages. Even when compared to other declarative programming languages like Haskell, Prolog offers greater expressiveness and syntactical flexibility, and therefore provides higher chances to define a domain-specific language internally by just extending its built-in operator table appropriately.

1.2. Goals and Addressed Problems

The subject of this thesis is the discussion of two approaches to implement domain-specific languages in Prolog: externally by defining a parser using DCGs, or internally by extending Prolog's operator table by appropriate operator definitions. We develop general criteria to decide which approach fits better for an already given or newly created DSL. Since these criteria can be applied using static analysis of example sentences, we provide a system that can answer whether given example sentences are a valid subset of Prolog. If so, it is possible to define the DSL internally. In this

case, required operator definitions are automatically inferred. As a result, we answer the question "Is it *possible* to model these example sentences as valid Prolog terms, and *how*?" theoretically as well as by an assisting tool. We discuss several limitations of the programming language Prolog to define DSLs internally, and how to overcome them by enhancing and modifying the syntax of the Prolog programming language as defined in the ISO Prolog standard. Most of these syntactical enhancements are backwards compatible to existing Prolog systems, others require only minor changes of constants that could be set by program flags.

For the definition of external DSLs, we simplify the process of writing and debugging grammars, parsers, and interpreters in Prolog, and how to intertwine them. Since all these steps typically work on an (abstract) syntax tree as their shared data structure, we extend the classical compilation scheme of DCGs to implicitly create a corresponding parsing tree.

Finally, the objectives to define DSLs externally or internally are applied to popular languages that are to be connected with Prolog, namely the extended Backus–Naur form (EBNF), GraphQL, and XPath. We expand our considerations about internal and external Prolog DSLs to a newly defined domain-specific language to express expert knowledge in the form of declarative if-then rules. This DSL has been applied in the field of change management in organisational psychology to build knowledge bases.

1.3. Main Results

In this section, we briefly summarise the main results of this thesis.

Discussion on the Two Approaches. With DCGs, there are few limitations to domain-specific languages that can be parsed in the traditional way with grammars. Because the set of external DSLs is a superset of internal DSLs, our discussion on whether implementing a DSL externally or internally is mainly focussed on the applicability of Prolog operator definitions to describe the language, resulting in an internal DSL.

Extensions to the Programming Language Prolog. When discussing internal DSLs, we present several parts of the ISO Prolog standard where the language definition could be enhanced to support more domain-specific languages. In particular, nearly all sentences from other programming languages already form a list of valid Prolog tokens – simply put, words starting with an uppercase letter could be

handled as variables, and all others as atoms. Most problems therefore arise only when these tokens are combined into valid Prolog terms.

We discuss extensions of the ISO Prolog standard to allow a more flexible syntax, while still being able to unambiguously parse the given code. For all of these syntax modifications, we discuss whether they are backwards compatible or could be set as a program-wide flag. This approach has already been used in existing Prolog implementations to provide better compatibility with external systems, e. g., in SWI-Prolog [136, Sec. 5].

Assisting Tools for the Definition and Usage of External DSLs. Though DCGs have been subject of research since 1980 [96], the tools to assist programmers in the process of defining grammars are still rare and limited. As part of our work, we present an interactive, web-based tool to visualise the execution of a DCG.

The main contribution of this thesis, in respect of defining parsers in Prolog to add support of external DSLs, is *library(dcg4pt)*. Its main features can be summarised as follows:

- Grammars specified as DCGs in Prolog are source-to-source transformed into traditional DCGs that store an additional argument that holds the corresponding syntax tree. Since our tool allows and implements modifications of the syntax tree, we use the term *parse tree* in this thesis instead, which covers both concrete and abstract syntax trees as well as intermediate hierarchical terms that represent a program. The name *library(dcg4pt)* stands for *definite clause grammars for parse trees*.
- *library(dcg4pt)* can be used as a drop-in replacement for existing DCGs, i. e., no further adjustments on the grammar have to be made.
- Grammars of computer languages often make use of sequences and optional nonterminals. Our library extends the notation of DCGs by meta-nonterminals to add support for sequences of nonterminals as well as symbols of any arity. This idiom can also be used to denote an element as optional.
- *library(dcg4pt)* has been developed with a focus on logical purity. It can be used both for parsing and serialisation, i. e., to create a parse tree by a given source code, but also the other way round to serialise a given parse tree back to a string or file. Although in this thesis we focus on the first direction, *library(dcg4pt)* has been applied by the author in [82] to create a feature-rich linter and code formatter for Prolog programs that ensures the adherence to common coding guidelines.

Automatic Operator Inference for Internal DSLs. In existing Prolog systems, an operator's definition has to be given before it can be used. In general, these operators are built-in, provided by an included library, or specified in the executed source code. If not given in advance, the program cannot be parsed. To the best of our knowledge, there is no Prolog system that relaxes this condition and is able to parse a Prolog term with some of its contained operators not yet being present in the operator table.

Therefore, we develop and present *library(plammar)*. It is a flexible and by default strictly standard-compliant parser for Prolog source code, written in Prolog. *library(plammar)* is able to infer possible operator definitions based on examples. Its main features can be summarised as follows:

- The library is used for static analysis of Prolog source code. It provides predicates to parse code snippets into a list of tokens, and transform it into a concrete or abstract syntax tree.
- Our library supports several Prolog dialects and can therefore handle source code that was originally written for SWI-Prolog as well as for GNU Prolog, or any other Prolog system that respects the ISO Prolog standard. Language features that are not (yet) part of the standard can be finely tuned, and separately enabled or disabled.[5] As a consequence, *library(plammar)* allows to give newly discussed language features and syntax extensions a shot.
- Similar to *library(dcg4pt)*, *library(plammar)* can be used both for parsing and serialisation. This way, it forms the basis for expressive Prolog source code transformations, which we emphasise as future work when concluding this thesis.

Exemplary Applications for EBNF, GraphQL, and XPath. We apply both approaches to define DSLs for real-life computer languages. We add support for GraphQL in Prolog using both the classical approach to define a parser with DCGs, as well as by extending the Prolog programming language. In addition, we present how to define XPath as an internal Prolog DSL, and discuss the required changes to the syntax of Prolog to use GraphQL as an internal DSL, too. As an introductory example, we refer to our work done in [109, 110, 126], and implement a simple language to model knowledge in the field of change management in the form of *if–then* rules.

[5] For instance, SWI-Prolog allows to use digit groups in large integers in the form of `10 000` and `10_000` instead of only writing `10000` [136, Sec. 2.16.1.5]. This deviation from the ISO Prolog standard can be deactivated only by using the more general `iso(true/false)` flag, which also includes further changes. Our contribution *library(plammar)* provides the options `allow_digit_groups_with_space(true/false)` and `allow_digit_groups_with_underscore(true/false)` to separately enable and disable this SWI-Prolog-specific language feature.

Since it lays the foundation for our implementation of a fully-featured Prolog parser, we discuss how to define context-free grammars given in EBNF as an internal DSL in Prolog. This way, parts of Prolog's syntax rules as given in the ISO Prolog standard can be directly used as Prolog source code. The resulting Prolog program allows to parse and to serialise Prolog code.

1.4. Thesis Structure

The overall objective of the thesis is to push the state of the art when defining and using DSLs in Prolog, both internally and externally. With *library(plammar)*, we provide a tool that assists with the decision on which technique to use. In addition, we discuss and present ways and tools to simplify the implementation of parsers for external DSLs, and the inference of appropriate operator definitions for internal DSLs. The actual contents and contributions of the chapters of this thesis are presented below.

1.4.1. Overview and Objectives per Chapter

The document is organised into eleven chapters. After Chapter 1, which gives an introduction to the research problems and constitutes the motivation for this thesis, we continue with Chapters 2 to 4 on the foundations of the logic programming paradigm, Prolog, and its integration as a DSL into other programming languages.

- Chapter 2 gives an orientation about the context of the thesis at hand. We present the general idea of declarative programming and logic-based programming languages. Since those share first-order logic as their mutual foundation, we introduce its basic concepts and notions. Most of the terms introduced in this chapter are reconsidered again in the context of Prolog later throughout this work.
- Chapter 3 starts with a short introduction to Prolog and its foundations. Its syntax, as defined in the ISO Prolog standard, is presented in more detail, since it is the basis for the Prolog parser and serialiser that is described later in Chapter 9. Chapter 3 also introduces more advanced Prolog idioms and libraries which are either not defined in the ISO Prolog standard or not part of all major Prolog systems, but used in several subsequent chapters.
- Chapter 4 first introduces DSLs in general, and gives an overview of the ways to define and use them. The chapter concludes with a detailed survey of existing solutions to integrate Prolog as a DSL into other programming languages,

which is the opposite direction of what is discussed in this thesis but often raises similar research questions.

Chapters 5 and 6 present the two opposite approaches to integrate DSLs, either internally or externally, into Prolog.

- Chapter 5 provides an overview of the possibilities Prolog offers to mimic a DSL. It includes a detailed introduction to Prolog's parentheses-free syntax based on user-defined operators. We introduce two approaches to further process an internally specified DSL – either by term expansions, or with the help of a meta-interpreter. As two practically used examples, we define if-then rules and EBNF as internal Prolog DSLs.

- In Chapter 6, we present how to define languages externally with the help of definite clause grammars, and integrate them next to Prolog source code with quasi-quotations. Since DCGs constitute an internal DSL to describe grammars, the aforementioned considerations are adapted to their processing. For comparison, Chapter 6 reconsiders if-then rules again and defines them as an external DSL. As an example application, we present *library(graphql)*, an external query language for deductive databases.

Though this external approach looks straight-forward, its current state of the art is verbose and error-prone. In Chapters 7 and 8 we therefore present two tools that assist with the definition of and work with external DSLs and DCGs.

- Chapter 7 introduces a web-based tool that improves the work with definite clause grammars. The graphical tracer visualises the execution of a DSL's underlying DCG. We therefore discuss several techniques to interactively collect information about applied grammar rules.

- One of the main contributions of this thesis, *library(dcg4pt)*, is presented in Chapter 8. We extend the classical structural term expansion of DCGs in order to store an additional parse tree argument. With its support for meta-nonterminals to specify optional and sequences of nonterminals, it is well-suited to define the syntax of external DSLs in a way that automatically creates a corresponding parse tree, and still can be used for both parsing and serialisation.

An application is given with our Prolog parser *library(plammar)* in Chapter 9, which adopts the aforementioned considerations to treat Prolog like any other external DSL.

– In Chapter 9, we apply the classical approach to add support for a language using a lexer and parser to the programming language Prolog. As one of the main contributions, we provide *library(plammar)*, a Prolog-based parser and serialiser for Prolog. We present several improvements to achieve a reasonable parsing performance. After all, the techniques presented in this chapter also underline Prolog's built-in capabilities to define parsers, compilers, and interpreters for any domain-specific language.

With *library(plammar)*, Prolog source code can be analysed statically. This forges a bridge in Chapter 10 to the initial question of which extensions to the operator table and to the syntax of Prolog are required so that a given DSL can be integrated internally.

– In Chapter 10, we elaborate on *library(plammar)*'s capabilities to automatically infer possible operator definitions from given example code snippets. Such operator definitions are necessary to specify the internal DSL. In addition, several extensions to the syntax of Prolog as originally defined in the ISO Prolog standard are discussed. They can be divided in changes that are related to the lexer and those that define which tokens constitute valid Prolog terms and programs. We reconsider GraphQL again, which was formerly defined as an external DSL, and can be instead implemented internally by appropriate type definitions and some of the proposed Prolog language extensions. As a second example, we present the definition of XPath expressions as an internal DSL. It has been applied in a Prolog-based XML Schema validator.

The thesis finally concludes in Chapter 11 with a summary and an outline of potential future work and remaining challenges in the definition and usage of DSLs in Prolog.

Appendices. Appendix A lists all scientific contributions of the author during the thesis in a condensed form. In addition, we mention developed software that covers different topics. In Appendix B, we provide additional source code examples which we shortened in the chapters before. Appendix C contains predicate definitions that are not part of the ISO Prolog standard or shipped with all major Prolog system but referred to throughout the thesis. An index and description of all language extensions supported by *library(plammar)* is given in Appendix D.

1.4.2. How to Read this Thesis

It is the intention that this thesis is self-contained and its chapters are only loosely coupled. However, the implementation of *library(plammar)* as described in Chapter 9 treats Prolog as an external DSL, i. e., the methods that were introduced for external DSLs in Chapter 6 are then applied to parse Prolog on its own. In addition, it makes use of *library(dcg4pt)*, which is introduced in Chapter 8, and the definition of EBNF as an internal DSL from Chapter 5.

Though we shortly introduce the language Prolog and its history in Chapters 2 and 3, we assume the reader to have a basic knowledge of the fundamentals of Prolog. Required advanced language features that are not part of the ISO Prolog standard, or that are not included in the most popular Prolog systems (e. g., dicts and attributed variables), or that were added only recently to SWI-Prolog are introduced throughout the work at hand when needed.

Since our work is strongly related to the syntax of Prolog, we often refer to definitions of *Part I* of the ISO Prolog standard. For the sake of simplicity, we refer to them as Iso, with Iso 6.3 representing the syntax definitions given in [55, Sec. 6.3].

1.4.3. Source Code Examples and Prolog Predicates

The provided tools, in particular *library(dcg4pt)* and *library(plammar)*, are entirely written in Prolog and available as open-source packages in the list of add-ons for SWI-Prolog. In this thesis, we include several examples of concrete Prolog source code of these libraries, as well as exemplary applications in Prolog's toplevel. This should allow readers to follow our considerations and try out actual executable code snippets. In our opinion, this is in particular useful for tracing the execution of the programs we present. The same applies for term expansions, which are otherwise often a cause for obfuscation, due to their nature of changing compiled code in-place.

Besides Prolog as the targeted host language, this thesis includes several other source code fragments, in particular when discussing various domain-specific languages that are aimed to be integrated. To clearly indicate the currently discussed language, all code listings contain a flag in the top-right corner with an appropriate language indicator or abbreviation. Only if code contains language extensions that are specific to SWI-Prolog and otherwise incompatible, we use SWI-Prolog instead of the more general Prolog flag. The source code examples in Chapter 10 do not carry a flag that indicates their depicted language, because they form valid Prolog programs only if the discussed extension is implemented and enabled. To distinguish

Prolog queries that are executed from actual Prolog source code, we mark them by the TOPLEVEL flag. The computed answer might be shortened and reformatted; alternative solutions computed via backtracking are indicated by the `;` symbol.

References to source code elements (e. g., for predicates or operators) in the running text are written in `typewriter` font; their programming language should appear by the context. Source code symbols which might be otherwise confused with punctuation marks, as well as longer code fragments are highlighted by a `grey background`. In case of facts, rules, and toplevel goals, we omit the trailing full stop `.` in this notation for the sake of readability, e. g., in `?- halt`. Placeholders in source code elements in the running text are furthermore printed in cursive characters if they strictly need to be replaced, because no free variable is allowed at their position. For instance, in the functor name of an abstract predicate *`Tag`*`(Arg)` the variable `Tag` has to be bound, since the compound is otherwise an invalid Prolog term.

When referring to Prolog predicates and operators, we always specify their full functor, i. e., the functor name together with its arity, separated by `/`. In addition, we indicate Prolog predicates and operators which are part of the ISO Prolog standard by a subscripted "ISO" (e. g., `clause/2`$_{\text{ISO}}$), and those that are shipped with SWI-Prolog similarly by "SWI" (e. g., `dict_pairs/3`$_{\text{SWI}}$). Non-standard predicates, operators, and program flags for which we provide a description and their implementation, are annotated by a reference to their definition in the corresponding section, Appendix B, or Appendix C in square brackets instead, e. g., in `otherwise/0`[C.7]. Predicate functors and operators without a subscripted text are defined in the section they appear in. Nonterminals intended to be used or defined by definite clause grammars are denoted by `//` instead of `/`, e. g., `integer//0`. We always use this full notation for nonterminals even in case of an arity of 0 in order to avoid confusion with terms of the same name, e. g., when discussing the nonterminal's corresponding parse tree.

1.5. Contributions

During this thesis, multiple works closely related to the discussed subject have been published in journals, conference proceedings, and as open-source software on GitHub. They are listed in this section. We group the contributions by their type and put them in context to their corresponding chapter of this thesis. The scientific contributions are also part of the Bibliography and cited in the chapters. A condensed version of this list is given in Appendix A, ordered by the publication's date. In publications with co-authors that highlight **Falco Nogatz**, the author of this thesis has contributed the majority of the content, including implementations.

1.5.1. Publications in Journals and Conference Proceedings

In [109, 110, 126], the general idea of defining an internal DSL has been discussed for rapid prototyping in Prolog. These publications describe its application for *if-then* rules to model knowledge in the field of organisational psychology. [109] is the extended journal version of the conference paper [110]. The if-then rules serve as a recurring, running example throughout the thesis. Compared to the aforementioned publications, we extend their definition as an internal Prolog DSL in Section 5.4, and expand the idea of their meta-interpretation and possible term expansions into plain old Prolog clauses.

> [126] Rüdiger von der Weth, Dietmar Seipel, Falco Nogatz, Katrin Schubach, Alexander Werner, and Franz Wortha. Modellierung von Handlungswissen aus fragmentiertem und heterogenem Rohdatenmaterial durch inkrementelle Verfeinerung in einem Regelbanksystem. *Psychologie des Alltagshandelns*, 9(2):33–48, 2016.

> [110] Dietmar Seipel, Rüdiger von der Weth, Salvador Abreu, Falco Nogatz, and Alexander Werner. Declarative Rules for Annotated Expert Knowledge in Change Management. In *5th Symposium on Languages, Applications, Technologies (SLATE 2016)*.

> [109] Dietmar Seipel, Falco Nogatz, and Salvador Abreu. Domain-Specific Languages in Prolog for Declarative Expert Knowledge in Rules and Ontologies. *Computer Languages, Systems & Structures (COMLAN)*, 51C:102–117, 2018.

Prolog has a long history of representing knowledge, either using a DSL or in the form of a controlled natural language (CNL). Combining both approaches, Prolog has proven to be suitable for defining new DSLs with a natural language flavour, like for the relational query language SQL. In the extended abstract [108], the knowledge representation in Prolog using these two approaches is discussed. This comparison is taken up again in Section 5.5 of the work at hand.

> [108] Dietmar Seipel, **Falco Nogatz**, and Salvador Abreu. Prolog for Expert Knowledge Using Domain-Specific and Controlled Natural Languages. In *8th Language & Technology Conference: Human Language Technologies as a Challenge for Computer Science and Linguistics (LTC 2017)*, pages 138–140, 2017.

CNLs are subject of recent research particularly because smart voice-controlled devices are becoming more present in daily life, with many different applications integrated into platforms like Amazon's conversational agent Alexa. In [83], we present a framework that assists with the development of skills for Amazon Alexa in Prolog.

> [83] **Falco Nogatz**, Julia Kübert, Dietmar Seipel, and Salvador Abreu. Alexa, how can I reason with Prolog? In *8th Symposium on Languages, Applications, Technologies (SLATE 2019)*, volume 74 of *OpenAccess Series in Informatics (OASIcs)*, pages 17:1–17:9, 2019.

In Chapter 7, we present an interactive, web-based visualisation for DCGs. It is based on the corresponding publication [81], but focusses on the discussion of how to collect the required DCG tracing information. In this regard, we treat DCGs as an internal Prolog DSL to describe grammars, and compare their application via modified term expansions, as well as via a modified tracing meta-interpreter with built-in tracing.

> [81] **Falco Nogatz**, Jona Kalkus, and Dietmar Seipel. Web-based Visualisation for Definite Clause Grammars using Prolog Meta-Interpreters: System Description. In *20th International Symposium on Principles and Practice of Declarative Programming (PPDP 2018)*, pages 25:1–25:10. ACM, 2018.

Our *library(dcg4pt)*, which is in detail presented in Chapter 8, has been first introduced in [85]. In this paper, we also propose to define EBNF as an internal DSL in order to make use of the syntax grammars specified in the ISO Prolog standard. Section 5.6 of this thesis expands this idea, which leads to our *library(plammar)* from Chapter 9.

> [85] **Falco Nogatz**, Dietmar Seipel, and Salvador Abreu. Definite Clause Grammars with Parse Trees: Extension for Prolog. In *8th Symposium on Languages, Applications, Technologies (SLATE 2019)*, volume 74 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:14, 2019.
> Honoured with the symposium's *Best Paper Award*.

An earlier version of the implementation of GraphQL as an external Prolog DSL has been published in [84]. This paper shortly discusses Prolog's limitations to define GraphQL internally as well, which is finally taken up again in the work at hand in Section 10.3.

> [84] **Falco Nogatz** and Dietmar Seipel. Implementing GraphQL as a Query Language for Deductive Databases in SWI-Prolog Using DCGs,

Quasi Quotations, and Dicts. In *Proc. 30th Workshop on (Constraint) Logic Programming (WLP 2016)*.

An application of the Prolog parser we present in Chapter 9 has been presented in the publication [82]. It makes use of our *library(plammar)* to analyse the adherence to coding style guidelines. The implemented linter is applied to several thousand source code files from SWI-Prolog's package ecosystem, and therefore serves as a real-world test for compliance to the ISO Prolog standard and the reasonable performance of the resulting Prolog parser. The empirical study results in optimisations for *library(plammar)* using a finite-state machine as presented in Section 9.3.

[82] Falco Nogatz, Philipp Körner, and Sebastian Krings. Prolog Coding Guidelines: Status and Tool Support. In *Technical Communications of the 35th International Conference on Logic Programming (ICLP 2019)*. The authors contributed equally. My focus was on the integration of *library(plammar)*, and the study on and support for Prolog extensions that are provided by SWI-Prolog, which forms Sections 3 and 4.2 of this article.

The definition of XPath as an internal DSL as presented in Section 10.4 is used to extend an XML Schema validator that is completely written in Prolog. It has been first presented in [80]. A revised version of this system description has been published in [79].

[80] **Falco Nogatz**, Jona Kalkus, and Dietmar Seipel. Declarative XML Schema Validation with SWI-Prolog: System Description. In *Proc. 31st Workshop on (Constraint) Logic Programming (WLP 2017)*.

[79] **Falco Nogatz** and Jona Kalkus. Declarative XML Schema Validation with SWI-Prolog. In Dietmar Seipel, Michael Hanus, and Salvador Abreu, editors, *Declarative Programming and Knowledge Management – Revised Selected Papers of Declare 2017*, pages 187–197, 2018.

The seamless integration of Prolog with other programming languages has been subject to research for years, though most solutions focus on the embedding of Prolog, i. e., treating Prolog as a domain-specific language. Since the general research questions are similar – for instance: *Is it possible to embed (a subset of) Prolog as an internal DSL into another language? Which changes to the syntax of Prolog are required to do so?* –, we present some of these existing systems in other programming languages in Section 4.3. One example is *CHR.js*, an interpreter and compiler which

extends JavaScript by Constraint Handling Rules (CHR). This constraint-logic programming language is shipped with all major Prolog systems. In [78, Sec. 5], we compare the means that Prolog and JavaScript provide to integrate the domain-specific language CHR into the host language JavaScript. It is the basis for Section 4.3.3 of this thesis.

> [78] **Falco Nogatz**, Thom Frühwirth, and Dietmar Seipel. CHR.js: A CHR Implementation in JavaScript. In *Rules and Reasoning (RuleML+RR 2018)*, volume 11092, pages 131–146. Springer, 2018.

1.5.2. Open-Source Software

Where possible, software components created as part of this thesis have been packed as a standalone library for SWI-Prolog. They have been published as part of SWI-Prolog's public package list,[6] and as open-source repositories on GitHub. For each software component we provide a short description (in *slanted font*), and information about the project's context and positioning with respect to this thesis. Additional software that was created by the author during the thesis but is not directly related to the domain of discourse is mentioned in Appendix A.3.

Chapter 8 describes how to adopt the classical term expansion for DCGs in order to store an additional argument that holds the parse tree. The corresponding implementation, *library(dcg4pt)*, can be used as a drop-in replacement for SWI-Prolog's built-in compilation of DCGs:

> *library(dcg4pt)*. https://github.com/fnogatz/dcg4pt, MIT License.
> *Extend definite clause grammars for Prolog by an additional argument to automatically store the parse tree.*

The Prolog parser, which we present in Chapter 9, has been bundled as *library(plammar)*, which is a portmanteau word for *Prolog grammar*. Inferring possible operator definitions from given example code snippets is just a small piece of its functionality. Besides parsing, it also supports the serialisation of a Prolog program based on its concrete or abstract syntax tree. In [82], *library(plammar)* is used to create abstract syntax trees from Prolog programs, and as a by-product checks the adherence to various Prolog coding style guidelines.

> *library(plammar)*. https://github.com/fnogatz/plammar, MIT License.
> *A Prolog grammar written in Prolog, for parsing and serialising Prolog code.*

[6] Packs (add-ons) for SWI-Prolog, https://www.swi-prolog.org/pack/list.

plammar-community-evaluation. https://github.com/fnogatz/plammar-community-evaluation, MIT License.
Data of the empirical study in [82].

The framework that assists with the development of skills for Amazon's conversational agent Alexa in Prolog as presented in [83] has not been bundled as a library for SWI-Prolog, as it needs to be adapted for each new skill. As an example application, we combine Amazon Alexa with the CNL Attempto Controlled English (ACE) and process commands with the help of the Prolog-based ACE Reasoner.

alexa.pl. http://github.com/fnogatz/alexa.pl, MIT License.
Amazon Alexa skill development with SWI-Prolog.
This framework used to create skills for Amazon Alexa was originally created as part of the bachelor's thesis of Julia Kübert [69].

library(race). https://github.com/fnogatz/race, MIT License.
Prolog client for the SOAP interface of the Attempto Reasoner RACE.

The XML Schema validator for SWI-Prolog, that motivates the definition of XPath as an internal DSL in Section 10.4 is published as *library(xsd)*. It makes use of *library(date_time)* to add support for date-based comparisons in XPath functions.

library(xsd). https://github.com/fnogatz/xsd, MIT License.
Validate an XML Document against an XML Schema in SWI-Prolog.
This tool has been developed as part of a practical course at the University of Würzburg, Germany. In addition to the author's work, it contains student contributions from Jona Kalkus, Kevin Jonscher, and Lucas Kinne.

library(date_time). https://github.com/fnogatz/date_time, MIT License.
Logical arithmetic on dates and times in Prolog.
This library is an adapted and improved version of the original *date_time* library that is part of Amzi! Core Components by Amzi!, Inc., to be compatible with SWI-Prolog's module system.

Most of these libraries have been developed in a test-driven way by the help of *library(tap)*. Their command line interfaces make use of *library(cli_table)* to print nicely formatted ASCII tables, e. g., to give a list of all possible operator definitions that *library(plammar)* has inferred.

library(tap). https://github.com/fnogatz/tap, MIT License.
Write tests with SWI-Prolog with the Test Anything Protocol (TAP).
Original project handed over from Michael Hendricks in February 2019.

library(cli_table). https://github.com/fnogatz/cli_table, MIT License.
Pretty Unicode tables for the command line with Prolog.

To test our libraries under different versions of SWI-Prolog, we have created *swivm*, the *SWI-Prolog version manager*. In *library(plammar)*, we implemented several Prolog language extensions that have been added to and are supported by different versions of SWI-Prolog. The corresponding settings have been tested with the help of *swivm*, as it allows to install and execute SWI-Prolog in all versions since SWI-Prolog 5, back from 2002. In addition, swivm allows to automatise the installation of multiple, coexistent installations of SWI-Prolog, which makes it a valuable part in workflows for continuous integration, delivery, and testing.

swivm. https://github.com/fnogatz/swivm, MIT License.
Bash script to manage multiple active SWI-Prolog versions.
It is a heavily modified fork of the popular Node.js version manager nvm.[7]

The interactive, web-based visualisation for DCGs, which we present in Chapter 7, is based on SWI-Prolog's built-in *Pengines* module.

DCG Visualiser. https://github.com/fnogatz/dcg-visualiser, MIT License.
A web-based visualisation for definite clause grammars.
This web application using SWI-Prolog's *Pengines* library was originally created as part of the master's thesis of Jona Kalkus [59].

Originally developed as part of the author's master's thesis in 2015, we implemented a JavaScript module that adds support for CHR. In Section 4.3.3, it serves as an example for the opposite direction than discussed in this thesis, as it seamlessly integrates the logic-based programming paradigm into the imperative scripting language JavaScript. Since 2015, CHR.js has been continuously maintained and further improved for the publication at *Rules and Reasoning* [78].

CHR.js. https://github.com/fnogatz/CHR.js, MIT License.
Compile and run Constraint Handling Rules in JavaScript.

Finally, we mention in Section 9.5 a web-based visualisation of the concrete and abstract syntax trees that are produced by *library(plammar)*. It is based on the

[7] Node Version Manager nvm, https://github.com/nvm-sh/nvm, MIT License.

AST Explorer by Felix Kling, https://github.com/fkling/astexplorer. Our fork at https://github.com/fnogatz/astexplorer contains adaptions that are required to display Prolog parse trees. This includes minor changes to allow the *AST Explorer* to request the displayed syntax tree as a JSON document from a remote server. This server component is based again on SWI-Prolog's *Pengines*, and bundled with our *library(plammar)* in its `server` directory.

2

Logic-Based Programming

Reasoning using rules and exceptions is an important feature of our everyday life.

— Bob Kowalski[8]

The evolution to more than 700 different programming languages as mentioned in Chapter 1 has been a long journey with many partings of the way. As a result, today's world of programming languages is big and diverse. Regarding their underlying programming paradigm, they can be roughly classified in two categories: *imperative* and *declarative*. Imperative programming puts emphasis on how to solve a given problem, while declarative programming languages describe the requirements that have to be fulfilled by a possible solution. In Prolog, the latter is done in the form of implications. The foundation to deduce knowledge in a closed world based on these implications is the theory of first-order logic. In this chapter, we shortly introduce it and the history of logic programming as its application.

The chapter is organised as follows. In Section 2.1, we first give a brief overview of the concept of declarative programming in general. This paradigm again can be partitioned into two major branches: *functional* and *logic* programming languages. The latter are usually based on first-order logic, for which we give an introduction in Section 2.2. Though there are various universal evaluation methods for logic-based programming languages, all of them are based on the substitution of logic variables in the given problem description. This process, which is called *unification*, is introduced in Section 2.3. In Section 2.4, we show the fundamentals of the top-down deduction strategy that is used by Prolog, the *SLD resolution*. Finally in Section 2.5, we provide the example logic program *append*. This predicate can be used to concatenate or split lists, and serves as a running example throughout this thesis.

[8] Quote from "Time to think like a computer" [68]. Robert Anthony 'Bob' Kowalski is an American logician and computer scientist, who spent most of his career in the United Kingdom. He developed the SLD resolution and the procedural interpretation of Horn clauses (cf. Section 2.4), which established the basis for the operational semantics of logic programming. Consequently, Kowalski was honoured as one of the *15 Pioneers of Logic Programming* by the ALP in 1997.

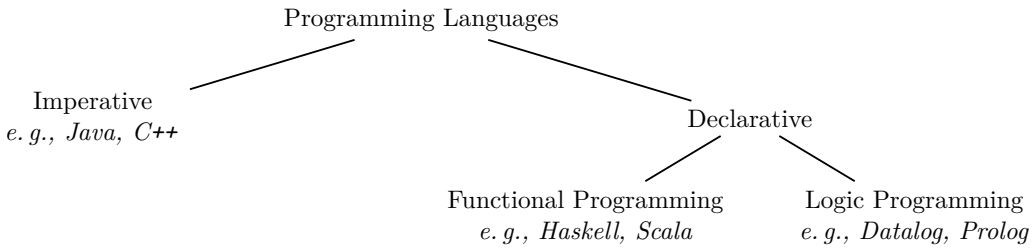

Figure 2.1: Programming paradigms and some of their representatives.

2.1. The Declarative Programming Paradigm

The world of programming languages can be classified in various ways. In this work, we do so using the program's area of application, with general-purpose languages on the one hand and domain-specific languages on the other. Another classification is based on the underlying mental model of programming. In the paradigm of *imperative* programming, developers express commands for the computer to perform, just like in the imperative mood in natural languages. Imperative programming focusses on the description of how the program should operate, effectively resulting in a set of instructions to change the program's state. In contrast, in the paradigm of *declarative* programming, one focusses on what the program should accomplish, without specifying the exact steps that have to be performed to achieve this.

Declarative languages can be further divided in *functional* and *logic* programming languages. Functional languages treat the computation as the evaluation of mathematical functions, whereas logic programming languages treat the computation as axioms and derivation rules. For instance, Haskell and Scala are popular functional programming languages; Datalog and Prolog are among the oldest yet most popular logic programming languages. Figure 2.1 illustrates the schema of different programming paradigms with some of their most popular representatives.

However, this classification based on programming paradigms is not strict. For instance, there is a trend in imperative programming languages to implement functional features. Recent versions of JavaScript ship with traditional functional constructs like `map` and `fold`; external libraries on-top allow to write JavaScript code that looks similar to Scala or Haskell [36]. In addition, there are attempts to combine the benefits of logic-based languages with traditional imperative programming languages, e. g., in [29, 89]. In most cases, this is done because of the concise and precise syntax that comes with declarative language constructs – a `map` is simply easier to read and understand than a loop that adds complexity because of an additional loop variable, or break and continue statements.

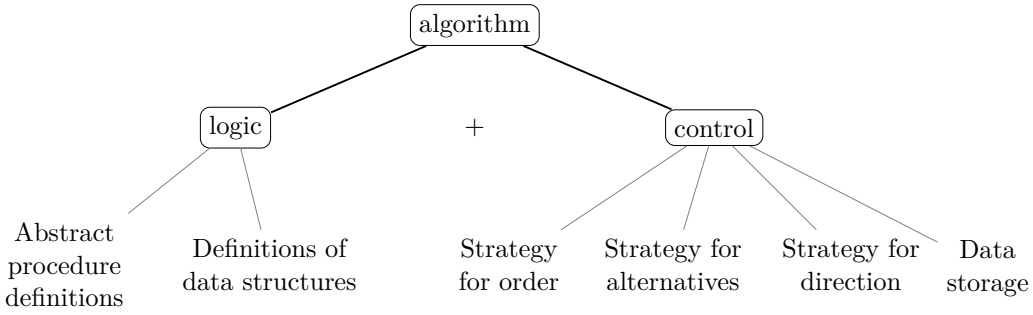

Figure 2.2: Components of algorithms.

The underlying programming paradigm also affects the program execution. In imperative programming languages, the program is described as a sequence of instructions. They are either directly compiled into low-level machine code (e. g., for C++), or executed one after another by a virtual machine (e. g., for Java). This way, the description of the problem (or of its solution, respectively) is encoded implicitly in the sequence of instructions, without a chance to clearly differentiate between the problem's characteristics, and the method used for its solution.

In contrast to this, in logic programming only the description of the problem is specified by the developer. The method for deducing an answer is universally defined in a generalised solver and thus separated from the problem description. This separation has been expressed by Robert 'Bob' Kowalski in the following well-known equation [67]:

$$algorithm = logic + control.$$

Here, *logic* indicates the description of the problem, including the definition of data structures and abstract procedures. On the other hand, *control* stands for the generalised evaluation mechanism to get a proper solution. There are various ways to implement the *control* component. These computation methods have to define strategies on how to handle multiple procedures (parallel or sequential?), alternatives (depth-first or breadth-first?), the solving direction (top-down or bottom-up?), and how to store their data.

This separation with its subquestions is depicted in Figure 2.2, which is a slightly adapted version of Kowalski's original work [67, p. 425]. Having a separate *logic* and *control* component has several advantages:

– Both parts can be developed independently, and may use different languages. In this regard, the *logic* can be specified in a DSL, while the *control* part can be implemented in some other language, e. g., C++, to ensure a fast execution. As a consequence, when describing the problem, it is not necessary to know

how the *control* part operates on this description; knowledge of the declarative reading of the problem specification suffices.

– Since the *control* component is universally defined, it works with any (sub-) set of problem descriptions. For an underdetermined problem description, the universal solver returns a partial solution. By specifying more constraints of the problem, a returned solution can then be refined step-by-step, resulting in an incremental solver.

– Changes to the *control* affect only the efficiency to find a solution. This allows to incrementally refine and improve the universal method that handles the problem description.

This separation of concerns – splitting the algorithm into a *logic* and a *control* component – is the reason why our intended host language Prolog is sometimes referred to as a domain-specific language tailored to applications of logic programming. While the way a problem is described (i. e., the *logic* part) is usually the same for all Prolog systems, their *control* component could be specified in various general-purpose languages.

When developing algorithms or software at scale, besides logic and control a third part is of great importance: their explanations and source code documentation. Similar to declarative programming, the *literate* programming paradigm as conceived by Donald Knuth shifts the programmer's focus from writing software in the manner and order imposed by the computer to develop them in the order demanded by the logic instead [64]. Systems following this concept allow to arbitrarily mix comments and source code, and the program's execution is independent from the literate order of source code snippets. The program's code is then origin for both the machine executable code and its documentation.

In this regard, logic programming languages are suitable also for the literate programming paradigm – because of the inherent separation, the *control* component could be implemented in a way that it does not rely on the source code order. In addition, logic programming languages are usually *small-syntax languages* with only minor syntax restrictions, leaving a lot of freedom to define new means for embedding comments, which thus can be closely interwoven with the executable source code. First attempts have been made with SWI-Prolog's *library(pldoc)* [131].

There are various approaches to implement the *control* component of logic programming languages. Besides Prolog, there are also other logic programming languages like Datalog, which usually differ in their syntax and procedural semantics. But with all being representatives of the logic-based programming paradigm, these languages

have in common that their *logic* component is usually based on first-order logic, for which we give a short introduction in the next section.

2.2. First-Order Logic as the Basis for Logic Programs

First-order logic is an extension of propositional logic. In addition to declarative propositions, it allows the use of quantified variables, functions, and predicates. It is therefore also called *predicate logic*, *predicate calculus*, or *quantificational logic*. The term *first-order* is used to distinguish the calculus from the *zeroth-order* propositional logic on the one hand, and *higher-order* logic on the other [4]. The latter additionally allows predicates and functions to have predicates as arguments, or in which one or both of predicates or functions can be quantified ("there is a predicate so that…", and similar propositions).

While in propositional logic variables are either true ($\top$) or false ($\bot$), in first-order logic they are entities in the domain $\mathcal{D}$ of discourse, which we call the *universe*. *Predicates* take entities of $\mathcal{D}$ to define their relations; though predicates with zero or one argument are also possible.

The *semantics* of first-order logic is described in Sections 2.3 and 2.4. In this section, we first introduce its *syntax*. Similar to the specification of a programming language, the syntax describes the finite sequences of symbols that are legal expressions in first-order logic.

Alphabet. The alphabet is a countably infinite set of symbols that can be used to form an expression. Symbols are either *logical* or *non-logical*. Logical symbols have always the same meaning and are not dependent on the domain of discourse. First of all, they include the logical symbols known from propositional logic:

- truth constants for true and false, usually denoted as $\top$ and $\bot$,
- logical connectives like $\wedge$ for conjunction, $\vee$ for disjunction, $\rightarrow$ for implication, $\leftrightarrow$ for equivalence, and the unary operator $\neg$ for negation,
- an infinite set $\mathcal{V}$ of variables, which we denote by uppercase letters at the end of the alphabet, i. e., $\mathcal{V} = \{ X, Y, Z, \ldots \}$, and
- punctuation symbols like parentheses and brackets.

Secondly, as an extension to propositional logic, first-order logic also includes the quantifier symbols $\forall$ for *universal quantification* and $\exists$ for *existential quantification*. For more descriptive variable identifiers, longer names starting by an uppercase letter might be used.

Another extension to the alphabet known from propositional logic are non-logical symbols. They represent predicates (relations) and functions on the domain of discourse, and are therefore specific to the described problem. From a syntactical point of view, predicates and functions are similar: they consist of a predicate symbol or function symbol, and an arbitrary number of arguments. We denote predicate symbols by lowercase letters from the end of the alphabet $\Pi = \{ p, q, r, \dots \}$, whereas function symbols are denoted by the leading lowercase letters $\Sigma = \{ a, b, \dots, f, g, h, \dots \}$. For more descriptive identifiers, longer symbols starting by a lowercase letter might also be used. Π and Σ are the *predicate signature* or *function signature*, respectively. Together, they form the set of non-logical symbols, which are specific to the described problem and context.

In literature, the notation for elements of Π and $\mathcal{V}$ is sometimes vice versa, with lowercase letters used for predicate symbols, and uppercase letters used for variables. We use the formerly introduced notation, as it aligns with the syntax of Prolog.

Term. Given the alphabet of logical and non-logical symbols, we specify how to build complex terms and formulas. In first-order logic, a *term* is inductively defined. It has one of the following forms:

1. A *variable symbol* of $\mathcal{V} = \{ X, Y, Z, \dots \}$. Unlike propositional variables (which can either be $\top$ or $\bot$), they can represent any entity in the domain of discourse.
2. A *compound term* of the form $f(t_1, \dots, t_n)$, where f is a *function symbol*, and $t_1, \dots, t_n$ is the ordered set of terms as its *arguments*. The number of arguments is called *arity*. A compound term with an arity of 0, e. g., just a, is called *constant*.

The set of terms is denoted by $\mathcal{T}$. Only expressions which can be obtained by finitely many applications of this inductive definition are terms. In particular, no expression involving a predicate symbol is a term. Consequently, compound terms do not contain elements of Π as arguments, thus it is not possible to express propositions about predicates, which in contrast would be allowed in higher-order logic.

Besides *terms*, whose arguments are not further restricted, *ground terms* are terms that contain no variables. They are again inductively defined:

1. A constant is a ground term.

2. If f is a function symbol with an arity of $n \geq 1$, and $t_1, \ldots, t_n$ are ground terms, then $f(t_1, \ldots, t_n)$ is a ground term.

The set of all ground terms of a formula in first-order logic is called the *Herbrand universe*, named after the French mathematician Jacques Herbrand (1908–1931).

Formula. Formulas are typically denoted by uppercase letters $\{ F, G, \ldots \}$. Given terms over signatures (Σ, Π), a *formula* in first-order logic is inductively defined by the following rules:

1. If p is a predicate symbol of Π with an arity of $n \geq 0$ and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula. For $n = 0$ we write p.

2. If F is a formula, then $\neg F$ is a formula.

3. $\top$ and $\bot$ are formulas.

4. If F and G are formulas and $\oplus$ is a binary logical connective defined in the alphabet, then $F \oplus G$ is a formula.

5. If F is a formula and X is a variable of $\mathcal{V}$, then $\forall X\, F$ ("for all X, F holds") and $\exists X\, F$ ("there exists X such that F holds") are formulas.

Formulas built only from the first rule are called *atomic* formulas. A formula built only from rules 1 and 2 is called a *literal*. Ground atomic formulas are widely used to describe properties of a formula, e. g., interpretations and its models; literals are often used for proofs and derivations.

Free and Bound Variables in a Formula. In a formula, a variable may occur *free* or *bound*. Intuitively, a variable occurrence X is free in a formula F if it is not quantified by $\forall$ or $\exists$. With the given inductive definition of formulas, X occurs free in an atomic formula $F = p(t_1, \ldots, t_n)$ if and only if X occurs in F (rule 1). In complex formulas with logical connectives, X is free (or bound, respectively) if and only if X is free (or bound) in the operands (rules 2–4). In a quantified formula $\forall Y\, F$ or $\exists Y\, F$ (rule 5), X is free if and only if X is free in F and $X \neq Y$; X is bound if and only if $X = Y$ or X is bound in F.

A formula in first-order logic with no free variable occurrences is called a *first-order sentence*. In the following, we restrict ourselves to formulas of this form, since they have well-defined truth variables under an interpretation.

Horn Clause. The finite disjunction of literals $\{ L_1, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n \}$ with universally quantified variables from $\mathcal{V}$ in L_i is called a *clause*. Every formula given in first-order logic can be represented by a clause that is equisatisfiable, i. e., though it is not required to have the same model, it is satisfiable if and only if the original formula is satisfiable. An equisatisfiable clause to a given formula can be obtained using standard techniques like Skolemisation and equivalent transformations.

The special case of a clause with only a single positive literal ($m = 1$) is called *Horn clause*. They are named for the American mathematician Alfred Horn (1918–2001), who first pointed out their significance in 1951, and form the foundation of logic programming. With De Morgan's law, a Horn clause with variables $X_1, \ldots, X_p \in V$ can be equivalently written as a formula of the form

$$\forall X_1 \ldots \forall X_p \, (L_1 \leftarrow L_2 \wedge \cdots \wedge L_n).$$

Logic Program. Because after the Skolemisation all variables $X_1, \ldots, X_p \in V$ are bound and universally quantified, they are usually omitted. We call

$$L_1 \leftarrow L_2 \wedge \cdots \wedge L_n$$

with atomic formulas $L_1, \ldots, L_n$ a *definite clause*. The finite set of definite clauses and facts forms a *logic program*.

For $n = 2$ and $L_2 = \top$, the definite clause $L_1 \leftarrow \top$ is called a *fact* and is written as L_1 for short. Otherwise, the definite clause is called *rule*. L_1 is the *head* in both cases, and $L_2 \wedge \ldots \wedge L_n$ the *body* of a rule. Variables which occur only in the body are called *local variables*. Using logic equivalent transformations, the quantification of all local variables can be moved from the rule's head to its body, changing their universal quantifier $\forall$ to the existential quantifier $\exists$ due to the negation of the implication.

2.3. Theory of Unification

Following the definition of free and bound variables in formulas as introduced in Section 2.2, the same variable symbol X might occur simultaneously free and bound in different parts of a formula, e. g., in $G = p(f(X)) \wedge \forall X \, \neg p(X)$. In the first operand of the conjunction, X is free, while in the second it is universally quantified and therefore bound. We usually avoid such ambiguities by renaming the variables appropriately. This is done via substitutions.

Substitution. A *substitution* $\sigma : \mathcal{V} \to \mathcal{T}$ is a mapping from variables to terms, i. e.,

$$\sigma = \{ X_1 \mapsto t_1, \ldots, X_n \mapsto t_n \},$$

with pairwise different variables X_i. The mapping $X_i \mapsto t_i$ is called the *binding* of the variable X_i to the term t_i. We write X_i/t_i for short. A substitution with only ground terms t_i is called *ground substitution*. The empty (identity) substitution is denoted by ϵ.

Applying a substitution σ on a formula F is written in postfix notation as $F\sigma$. It means to simultaneously replace every occurrence of a variable X_i in F by the term t_i. The result $F\sigma$ is called an *instance* of that formula F.

It is $t_1\sigma = t_2\sigma$ if and only if t_1 and t_2 are syntactically equivalent over the Herbrand universe, i. e., with the containing free variables being bound over all ground terms. For an instance $F\sigma\sigma'$ with substitutions σ and σ', there is a substitution θ so that $F\sigma\sigma' = F\theta$. The substitution θ is called the *composition* of σ and σ', with $\theta = \sigma\sigma'$.

In our exemplary formula $G = p(f(X)) \wedge \forall X \neg p(X)$, the first variable occurrence of X can be renamed by $\sigma = \{X/Y\}$ and the instance $p(f(X))\,\sigma = p(f(Y))$. $G' = p(f(Y)) \wedge \forall X \neg p(X)$ eliminates the ambiguous usage of the variable name X. Though G and G' are not equivalent as they have different models due to the renamed variable, G is satisfiable if and only if G' is satisfiable.

Unification. In the previous paragraph, we have shown how to apply a given substitution to a formula. A common task in first-order logic is to find a substitution σ so that for two given literals t_1 and t_2 the equation $t_1\sigma = t_2\sigma$ holds. This substitution σ is called *unifier*, the process of finding it is called *unification*.

For instance, given a is a constant in the Herbrand universe, two literals $t_1 = p(f(Y))$ and $t_2 = p(X)$ can be unified by the substitution $\sigma = \{ X/f(a), Y/a \}$, with $t_1\sigma = t_2\sigma = p(f(a))$.

Most General Unifier. For the previous example, we easily find additional substitutions that satisfy the equation $t_1\sigma = t_2\sigma$, e. g., $\sigma' = \{ X/f(f(a)), Y/f(a) \}$, or $\sigma'' = \{ X/f(Y) \}$. For two given literals there is either no unifier or infinitely many. For instance, for $t_1 = p(X)$ and $t_2 = p(a)$, we could add additional variables and renamings, like in $\sigma'' = \{ X/Y, Y/a, Z/a \}$.

We therefore introduce the notion of the *most general unifier* (MGU). A substitution θ is MGU of two literals t_1 and t_2 if and only if $t_1\theta = t_2\theta$, and if for all

other unifiers σ, there is a substitution σ' so that $\sigma = \theta\sigma'$. In the previous example with $t_1 = p(X)$ and $t_2 = p(a)$, the MGU is $\theta = \{ X/a \}$. This MGU is unique, but there are also examples with various valid MGUs, e. g., for two literals $t_1 = p(X)$ and $t_2 = p(Y)$.

The *unification algorithm* [100] reports unsolvability or computes a complete and minimal singleton substitution of two given literals. For first-order unification, a MGU can be effectively computed in linear time [71, 94].

Occurs Check. As stated before, only expressions which can be obtained by finitely many applications of the inductive definition are terms. As a consequence, given two literals t_1 and t_2, their unifier is a finite substitution, i. e., a finite set of variables that are mapped to terms. Nevertheless, this could still result in infinite terms once the application of the unifier is not idempotent. For instance, two literals $t_1 = p(f(X))$ and $t_2 = p(X)$ have the MGU $\theta = \{ X/f(X) \}$, leading to the infinite term $t_1\theta^n = t_2\theta^n = p(f^n(X))$. To avoid infinite terms as solutions, the unification algorithm sometimes contains an additional check that in the computed variable substitution X_i/t_i with a compound term t_i, t_i does not include the variable X_i again. This additional check, called *occurs check*, has a great effect on the performance of the unification algorithm. It is therefore omitted by default in most Prolog systems. By not performing the occurs check, the worst case complexity of unifying a term t_1 with a term t_2 is reduced in many cases from $\mathcal{O}(size(t_1) + size(t_2))$ to $\mathcal{O}(min(size(t_1), size(t_2)))$ [97]. In particular, the frequent case of variable-term unifications (i. e., $t_1 = X_i$) is performed in $\mathcal{O}(1)$.

2.4. Computation with Logic Programs

First-order logic can be seen as a formal and unified language to describe the requirements relevant to a solution of a given problem. In a logic program, these requirements are specified in the form of definite clauses and facts, which therefore resemble the syntax and data structures of a programming language. Even the language's semantics are well-defined by the declarative reading of the formula, with all solutions for the specified problem being members of the formula's model. However, it remains open how to concretely calculate this model, i. e., the procedural semantics of logic programs. In particular it is of interest if a given ground atomic formula $p(t_1, \ldots, t_n)$, the *goal*, is part of the model. In case of an atomic formula with free variables, the substitution σ should be computed so that $p(t_1, \ldots, t_n)\sigma$ is in the model.

There are two major approaches to this reasoning with logic programs in a goal-oriented way. In *top-down* problem solving, we reason backwards from the conclusion, repeatedly reducing goals to subgoals via definite clauses until all subgoals can be proven by facts. In *bottom-up* problem solving, beginning with the facts new assertions are repeatedly derived forward to the hypothesis, until eventually the original goal is solved directly by the derived assertions. There are several proof procedures for each approach. For instance, bottom-up computation can be achieved with the help of hyperresolution [100]. In this section, we introduce the foundations and some properties of the SLD resolution. It is the top-down proof procedure that Prolog's execution is based on.

2.4.1. Top-Down Depth-First Inference with SLD Resolution

Resolution is a rule of inference leading to a refutation theorem-proving technique. While propositional logic relies on a propositional variable X_i and its complementary $\neg X_i$, first-order logic uses the literal L_i and its complement modulo unification $\neg L_i\sigma$. Given two clauses

$$K_1 = L_{1,1} \vee \cdots \vee L_{1,n_1},$$
$$K_2 = L_{2,1} \vee \cdots \vee L_{2,n_2},$$

with $L_{1,i} = \neg L_{2,i}\sigma$ for a unifier σ, we can infer the formula

$$F = (K_1 \setminus L_{1,i} \wedge K_2 \setminus L_{2,i})\sigma,$$

if $K_1 \setminus L_{1,i}$ and $K_2 \setminus L_{2,i}$ have no variables in common. F is called the *resolvent* of the *premises* K_1 and K_2. Typically, the resolution rule is iteratively applied in a suitable way for proving that a given formula in first-order logic is unsatisfiable.

The general idea of resolution has been refined by Kowalski in 1974 [66] to calculate a unifier σ so that for a given atomic formula F, $F\sigma$ can be inferred from the logic program. The atomic formula F in question is called *query* and is usually indicated by a leading "$\leftarrow$", so it is written as $\leftarrow F$.

Calculating the unifier σ can be achieved by finding a sequence of resolvents $G_0, \ldots, G_n$ with appropriate unifiers $\sigma_1, \ldots, \sigma_{n-1}$ that ends in the empty clause $G_n = \square$, starting with the query $F = \neg G_0$. Then, the sequence $\sigma = \sigma_1\sigma_2 \ldots \sigma_{n-1}$ used in the refutation is called the *computed answer substitution* for the goal F in the logic program P. Typically, the computed answer substitution is restricted to the variables that occur in the goal F, since all other variables in σ were local variables in the applied definite clauses.

SLD resolution is short for *selective linear definite clause resolution*. This refers to the fact that the only literal resolved in a resolution step is one that is uniquely selected by a selection function. It is linear in the sense that the proof of refutation needed for the computed answer substitution can be achieved by a linear sequence of goals $G_0, \ldots, G_n$.

The presented approach implicitly defines a search tree of alternative computations, in which the initial goal clause is associated with the root of the tree. Since there might be multiple clauses in the logic program whose head K unifies with the selected literal of the goal, each alternative is represented in the search tree by an edge from the current goal. The child nodes are built from the goal clause obtained by the resolution step of the current goal and the chosen alternative. There are two kinds of leaves in this search tree: *success nodes* are those, whose resolution results in the empty clause; *failure nodes* on the other hand are those, for which there is no corresponding clause in P whose head unifies with the selected literal, i. e., this goal cannot be proven by the logic program P.

2.4.2. Nondeterminism and Backtracking

The SLD resolution introduces several sources of nondeterminism. Firstly, as a goal G_i could consist of multiple literals $L_{i,1}, \ldots, L_{i,n_i}$, it remains unclear which is used first for resolution (i. e., to find a clause in P with a complement modulo unification to this literal). This order is defined by the selection function s, which maps a goal G_i to the used literal $L_{i,k}$.

In the SLD resolution, the selection function s is defined to choose the literal that has been most recently introduced into the resolvent. In the simplest case, such a last-in-first-out (LIFO) selection function can be specified by the order in which literals are written. Given the LIFO stack represented as a string, we simply use the left-most written literal, and denote this particular selection function by s_l.

However, there is no restriction to s on the literal that can be selected. Therefore, the selection function used in the resolution step could be defined more sophisticated. Because it directly influences the steps needed to, e. g., find a refutation, the choice of the selection function s greatly influences the computational performance of the resolution. Prolog is based on the SLD resolution with its LIFO-based selection function s_l. Since the choice of s_l is particularly interesting for large combinatorial problems (e. g., in constraint satisfaction problems, cf. Section 10.1.4), Prolog's choice for the prioritisation of recently added subgoals can be manipulated by writing a proper meta-interpreter (cf. Section 5.3).

A second nondeterminism is introduced by the choice of the corresponding clause $K \setminus L$ in P, represented by the different edges in the search tree. This again influences the program's performance and completeness: if the search space contains infinite branches and the search strategy chooses these in preference to finite branches, the computation does not terminate. In Prolog, the program's clauses are applied in the order they appear in the source code; a meta-interpreter can opt for a different approach.

Besides the questions on how to select the first subgoal for the next resolution step and how to select a possible corresponding clause, a third characteristic is to whether traverse through the search tree of alternative computations in a breadth-first or depth-first manner. If it is asked for all possible solutions, both approaches are identical regarding the program's performance. With respect to completeness, depth-first computation again does not terminate for search spaces with infinite branches.

If only the first answer is of interest, depth-first computation generally yields faster possible solutions. Once the computation reaches a failure node in the search tree, i. e., the current goal cannot be proven, the next alternative clause in the parent's node is applied; if there is none remaining in the search tree, this node is again treated as a failure node, and so on. This process is called *backtracking* and is a major language feature of Prolog.

In case there is not a single success node in the SLD search tree for a query $p(X)$ – i. e., it is failing –, it is known that its negation $\neg p(X)$ is true. This is due to the *closed-world assumption*, which presumes that any real-world statement that is true is also true in the first-order logic formalisation, either given as fact or as it can be inferred by the SLD resolution. Conversely, what cannot be inferred by resolution is known to be false. It is an important premise for the reasoning with negations, which SLD resolution is based on.

Logic programs typically consist mostly only of positive literals in the rule's bodies. A negated literal $\neg p$ is usually only used as a condition to apply a rule, e. g., in $F = q(X) \leftarrow s(X) \wedge \neg p(X)$. Here, it is not possible to infer the negated goal $\neg p(X)$ with the help of only SLD resolution. However, because of the closed-world assumption it is known that $\neg p(X)$ is true if and only if $p(X)$ cannot be inferred, i. e., it is failing. This refutation technique for negated goals is therefore called *negation as failure*.

2.4.3. Variables for Parameter Passing and Return Values

In Section 2.2, we introduced the terms *free* and *bound* as properties of variables that occur in first-order logic formulas, either standalone or quantified by $\exists$ or $\forall$. Since logic programs are built from definite clauses, all variables in a formula are required to be either universally quantified, or existentially quantified in case of local variables, making this distinction obsolete.

Both terms are differently connoted in the context of computation. Here, a variable X is *free* if there is no corresponding substitution $\sigma_i = \{ X/t \}$ in the computed answer substitution σ, or t is a free variable in the same way. Conversely, the variable X is *bound* if it is substituted by a compound term t, including constants.

A variable cannot occur simultaneously free and bound in different parts of a formula, since variables with the same symbol represent the identical entity in the domain of discourse. This concept is fundamentally different to variables known from imperative programming languages like Java, which handle a variable as a memory address and thus allow subsequent assignments. In logic programs, once the binding of a variable is established in the computed answer substitution, it cannot be modified. Only by backtracking a variable binding can be resolved, because a variable might hold different values in the various branches of the SLD search tree.

If the variable X is bound to a compound term t which is not ground, i. e., it contains a free variable Y, the variable X is called *partially bound* or *partially instantiated*. Though the binding of X cannot be undone, the variable's value can be further determined in the same branch of the search tree by (partially) binding the included variable Y. In this way, the known value of a variable can be specified in greater detail throughout the computation of a logic program.

As a result of having variables which can be partially bound, there is no need to explicitly declare input arguments and return values as required when defining functions and methods in most imperative programming languages – logic variables can serve as both. This allows for a bidirectional parameter-passing mechanism, where input and output can switch places. Consequently, the same logic program often describes both directions of computation. This behaviour is limited only by the operational semantics, i. e., how the computation handles the presented causes of nondeterminism and if the search tree is traversed depth-first or breadth-first, and if there are no side effects.

2.5. Logic Program Example: *append*

As an introductory example, we consider the logic program P that describes the predicate $append(X, Y, Z)$ with the meaning: Z is the result of concatenating the lists X and Y together, i. e., joining the lists end to end. It is described using the formulas F_1 and F_2 in first-order logic as follows:

$$\begin{aligned} F_1 &: \forall Y\ append(nil, Y, Y) \\ F_2 &: \forall X \forall Y \forall Z \forall E \\ &\quad append(cons(E, X), Y, cons(E, Z)) \leftarrow append(X, Y, Z) \end{aligned}$$

Here, we use the constant symbol *nil* to denote an empty list, and the function $cons(E, X)$ ("constructs") to describe a list with the first element E and a remaining list of X. In addition, we use the constant symbols $\{ a, b, \dots \}$ to represent list elements, so the Herbrand universe is

$$\begin{aligned} HU(P) = \{ &nil, a, b, \dots, cons(nil, nil), cons(a, nil), \dots, \\ &cons(cons(nil, nil), nil), \dots \}. \end{aligned}$$

Note that the Herbrand universe simply contains all combinations of functions and constants, including those which are not intended but valid according to the logic program's declarative reading. For instance,

$$append(nil, cons(a, a), cons(a, a))$$

is element of the model of P because of F_1, though in practice we restrict ourselves to instances $cons(E, X)$ where X is a list, i. e., either *nil* or of the form $cons(\cdot, \cdot)$ again, in contrast to $X = a$ as used as the predicate's last two arguments in the previous example.

The formulas F_1 and F_2 are given in the form created by normalisation steps, as having created a prenex normal form, and performing Skolemisation afterwards. To assist with the understanding of F_2, we provide an equivalent formula F_2' where E is moved to the rule's body, and the local variables L and M are introduced to put the unification from the rule's head in its body instead:

$$\begin{aligned} F_2 \equiv F_2' \coloneqq\ & \forall X \forall Y \forall Z \\ & append(X, Y, Z) \leftarrow \exists E \exists L \exists M \\ & \quad X = cons(E, L) \wedge\ Z = cons(E, M)\ \wedge append(L, Y, Z) \end{aligned}$$

In the logical reading, "=" acts as predicate that is true if both arguments can be unified. In the formulas, we use the infix notation "$A = B$" for "$=(A, B)$".

Similar to the transformations of F_2, F_1 can be rewritten as F_1' by moving the unification of X and Z with *nil* and Y from the rule's head to the rule's body and thus making it explicit:

$$F_1 \equiv F_1' \quad := \quad \forall X \forall Y \forall Z \; append(X, Y, Z) \leftarrow X = nil \; \wedge \; Y = Z$$

In the logic program P, a model has to satisfy all formulas, i. e., F_1 (or F_1', respectively) and F_2 (F_2') implicitly represent a logical conjunction. Because F_1' and F_2' have an identical rule head, they can be combined into a single rule. Consequently, P can be equivalently modelled by the single formula $F' \equiv F_1' \wedge F_2'$:

$$\begin{aligned} F' \quad := \quad & \forall X \forall Y \forall Z \\ & append(X, Y, Z) \leftarrow \\ & \quad (\, X = nil \; \wedge \; Y = Z \,) \vee \\ & \quad (\, \exists E \exists L \exists M \\ & \quad\; X = cons(E, L) \; \wedge \; Z = cons(E, M) \; \wedge \; append(L, Y, Z) \,) \end{aligned}$$

We will refer to these alternative forms F_1', F_2', and F' later in this work when discussing possible implementations of P in Prolog and other logic-based systems.

2.5.1. SLD Resolution for All Solutions

In the previous section, we introduced $append(X, Y, Z)$ as a predicate that holds the result of concatenating the lists X and Y in the last argument Z. Given two lists $X = cons(a, nil)$ and $Y = cons(b, nil)$, the SLD resolution should return the computed answer substitution $\{\, Z/cons(a, cons(b, nil)) \,\}$. It is the only success node in the SLD search tree for this query.

However, as stated in Section 2.4.3, there is no difference in input and output in logic programs. An alternative declarative reading of P is that X and Y are the result of splitting the ordered list Z at any position in-between. Thus, for a goal with only Z being bound, the SLD resolution returns appropriate substitutions for X and Y. This is the inverse function of the original meaning, i. e., the predicate *append* can be used to produce all combinations of two lists that together form the list Z.

Figure 2.3 shows the SLD search tree for an example query:

$$\leftarrow append(X_0, Y_0, cons(a, cons(b, nil)))$$

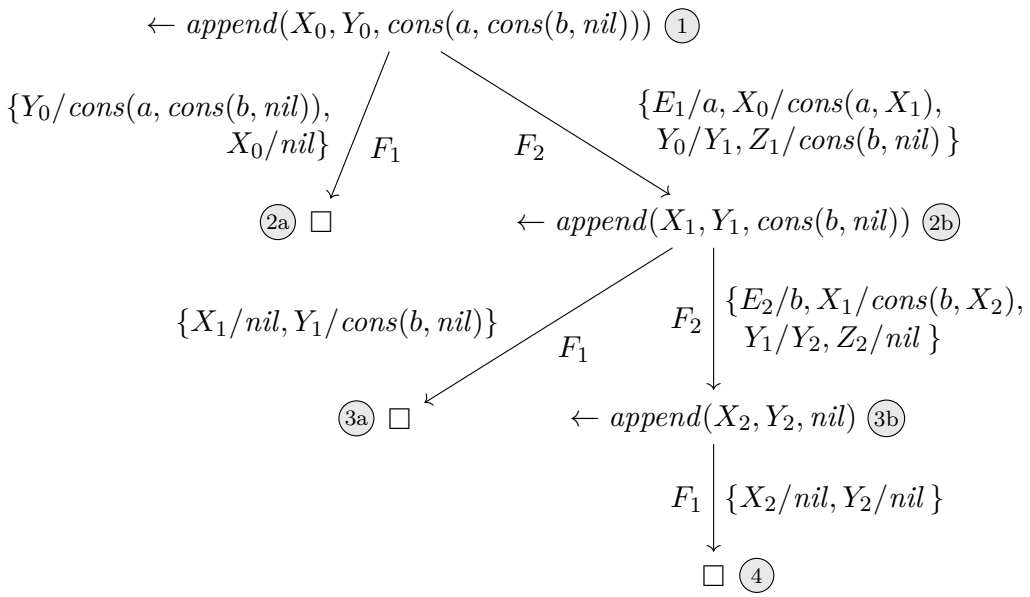

Figure 2.3: SLD search tree for the query $\leftarrow append(X_0, Y_0, cons(a, cons(b, nil)))$.

Simply put, this query asks for all pairs (X_0, Y_0) of lists so that concatenating them results in the list that holds only the constants a and b.

In the SLD search tree, nodes depict goals. The goals are indicated by a leading "←" again. The empty clause is denoted by □ and forms a success node. Each edge in the tree represents a performed resolution step. The edges are annotated by the used formula of P under the given substitutions. Variable symbols are enumerated, with X_0 and Y_0 being the free variables from the query, and X_i, Y_i, Z_i and E_i with $i > 0$ newly introduced local variables originating from the rule's body.

The circled numbers in Figure 2.3 indicate the order in which the computation of all three possible answer substitutions is performed by the SLD resolution:

(1) The search tree's root is formed by the initial query. It is used to find clauses and facts in program P, it can be unified with.

(2a) The query can be unified with the fact F_1 under the substitution $\{Y_0/cons(a, cons(b, nil)), X_0/nil\}$, which forms the computed answer substitution, as no local variables have been introduced. With this substitution, the current subgoal is inferred, and the computation continues with the next subgoal of (1). Since it consists only of a single literal, (2a) ends in a success node with the empty goal.

(2b) To search for additional solutions, (2a) backtracks to (1), trying alternative unifications for the initial query. It can also be unified with the rule F_2 under the substitution $\{E_1/a, X_0/cons(a, X_1), Y_0/Y_1, Z_1/cons(b, nil)\}$. The new goal

is the rule's body under this substitution, i. e., $\leftarrow append(X_1, Y_1, cons(b, nil))$. It depicts a sub-problem of the initial question, asking for all pairs (X_1, Y_1) of lists that together form the list which contains only b.

(3a) This goal again can be unified with the fact F_1 under the substitution $\{X_1/nil, Y_1/cons(b, nil)\}$. Together with the substitution of (2b), this results in the computed answer substitution $\{X_0/cons(a, nil), Y_0/cons(b, nil)\}$.

(3b) To search for additional solutions, the application of F_1 is backtracked, and F_2 is used for the goal of (2b) instead. The new goal $\leftarrow append(X_2, Y_2, nil)$ asks for all pairs (X_2, Y_2) which together form the empty list nil.

(4) There is only a single solution for this sub-problem: X_2 and Y_2 both have to be empty lists. This case is handled by the fact F_1, which unifies with the goal of (3b), resulting in the third computed answer substitution $\{X_0/cons(a, cons(b, nil)), Y_0/nil\}$. The goal, on the contrary, cannot be unified with the head of F_2, because this rule requires the third argument of *append* to be of the form $cons(\cdot, \cdot)$.

The search for alternative solutions terminates, as there is no other path in the SLD search tree. This is established by consecutively backtrack from (4) to the root node (1), where no step allows alternative clauses to resolve the currently examined goal with. In conclusion, there are only three ways to split the list of a and b, which all get successfully computed in the success nodes (2a), (3a), and (4).

2.5.2. Linear Refutation for a Particular Solution

As introduced in Section 2.4.1, the term "SLD resolution" refers to the fact that the proof of refutation for a single computed answer substitution can be achieved by a linear sequence of goals. It is the path of nodes from the root with the initial query to a particular success node in the SLD search tree. Figure 2.4 visualises this successful refutation for the answer substitution computed in (4) of Figure 2.3. Here, the linear sequence of goals is:

(1) $\leftarrow append(X_0, Y_0, cons(a, cons(b, nil)))$
(2b) $\leftarrow append(X_1, Y_1, cons(b, nil))$
(3b) $\leftarrow append(X_2, Y_2, nil)$
(4) $\square$

The graph in Figure 2.4 uses the same notation and step numbers as introduced for Figure 2.3. In addition, the edges are labelled by the partially bound values of both variables in question, X_0 and Y_0. For instance, in (2b) it is known that X_0 is

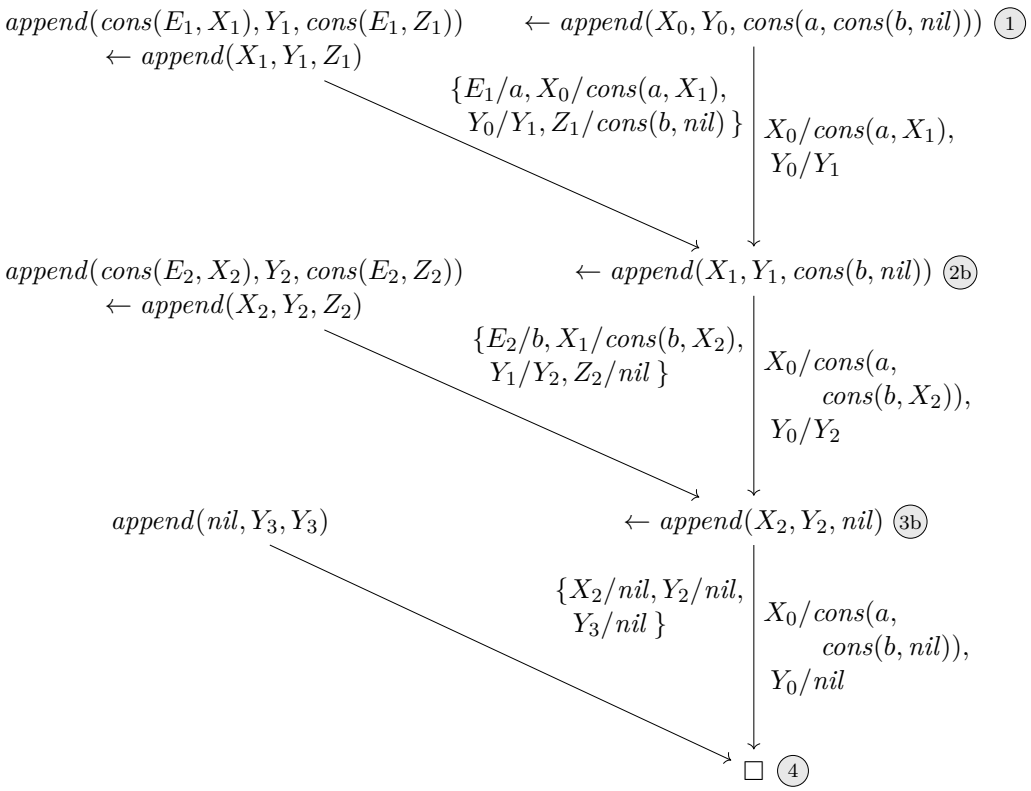

Figure 2.4: A linear refutation for the query $\leftarrow append(X_0, Y_0, cons(a, cons(b, nil)))$ that results in the computed answer substitution $\{\, X_0/cons(a, cons(b, nil)), Y_0/nil \,\}$.

the term $cons(a, cons(b, X_2))$ with a free variable X_2, and Y_0 is unified with the free variable Y_2. Both X_2 and Y_2 are finally bound to nil in (4), resulting in bindings to ground terms for X_0 and Y_0.

3

Programming in Prolog

1972 — Alain Colmerauer designs the logic language Prolog. His goal is to create a language with the intelligence of a two-year-old. He proves he has reached his goal by showing a Prolog session that says "No." to every query.

— A Brief, Incomplete, and Mostly Wrong History of Programming Languages[9]

The formalism of first-order logic as introduced in Chapter 2 is the basis for various logic programming languages. Their syntax is often quite similar and differs only in language-specific notations for the alphabet, i. e., how to denote logical and non-logical symbols. These differences range from varying source code symbols for truth values and logical connectives to conventions on how to denote variables, predicates, and functions.

In contrast, the semantics of the various logic programming languages is diverse. This is a result of the strict separation of the *control* component we presented in Section 2.1 – the actual computation is detached from the problem's description and thus can be performed in various ways. First of all, the formulas in first-order logic can be processed in a top-down or bottom-up manner, i. e., either starting from the user's query, or from all the facts in the knowledge base. But even then, the theoretic ideas of a particular computation method leave several open questions, like how to handle the various sources of nondeterminism in the case of the SLD resolution. All these definitions regarding the syntax and operational semantics together constitute a programming language.

Prolog was one of the first logic programming languages with its roots in the first-order logic. It was developed and implemented in Marseille, France, in 1972 by Alain

[9] Quote by Scala software engineer James Iry from the blog post "A Brief, Incomplete, and Mostly Wrong History of Programming Languages" (7 May 2009), available from http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html. Alain Colmerauer (1941–2017) was a French computer scientist. Being the creator of the programming language Prolog, he is one of the *15 Pioneers of Logic Programming* as honoured by the ALP in 1997. In addition, he was one of the main founders of the field of constraint-logic programming with the development of the *Prolog III* system.

Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses (cf. Section 2.4.1). The name originates from *programmation en logique*, French for *programming in logic*. In this chapter, we shortly introduce the basic concepts of programming in Prolog. For a more detailed introduction to Prolog we refer to [10, 20]; for more advanced Prolog usage we recommend [26, 86].

Terminology related to Prolog can be confusing, as unlike most other programming languages (e. g., PHP), Prolog has no canonical implementation, but instead many established systems, often with incompatible language extensions. The basic syntax of Prolog is specified in the ISO Prolog standard [55]. In the work at hand, we refer to it simply as *Prolog*. Actual implementations, e. g., *SWI-Prolog* and *GNU Prolog*, are given explicitly. Only where needed for differentiation, we provide specific versions of these implementations.

The introduction to Prolog in this chapter's first parts roughly follows the order of its theoretical counterpart in Chapter 2. The definitions get lifted from first-order logic to Prolog, with considerations regarding the language's syntax in Section 3.1 and its semantics and the underlying computational model in Section 3.2. This section also discusses the operator for unification in Prolog. In Section 3.3, we introduce common Prolog term structures, like strings, pairs, and lists. They are needed for the implementation of the exemplary first-order logic program of the predicate *append*, which is adapted for Prolog in Section 3.4. In Section 3.5, we introduce the Prolog idioms to examine, introspect, and modify the structure and behaviour of a loaded Prolog program. The clauses can be decomposed, analysed, and built just as normal Prolog terms, as we show in Section 3.6. This technique is particularly useful for creating dynamic Prolog programs, where clauses are modified at run-time. As a typical example, we present in Section 3.7 how to define Prolog predicates that mimic global variables. Finally in Section 3.8, we give a short introduction to SWI-Prolog's module system.

3.1. Writing Prolog Programs

In the same way we did for first-order logic in Section 2.2, we first describe the language's alphabet. It is the foundation to define the symbols that are legal expressions in Prolog. Because of its long history, Prolog's alphabet is traditionally composed only of the 7-bit US-ASCII character set. It consists of 95 printable characters for digits, lowercase and uppercase letters, punctuation marks, and a few miscellaneous symbols. The list of all printable characters is encoded in our implementation of `char_code/2` in Appendix C.5. Today's systems usually provide a larger alphabet. For instance, SWI-Prolog offers full Unicode support.

In addition to the printable characters, the ISO Prolog standard defines the seven *symbolic control characters* from ASCII code 7 up to 13. Among those, only the horizontal tab `\t` (9), line feed `\n` (10), and carriage return `\r` (13) are of practical relevance when writing Prolog programs. Unlike, for example, Haskell, the source code of Prolog is not whitespace-sensitive, i. e., the program's meaning does not rely on the indentation of the code. As a consequence, the symbolic control characters and the space character between symbols can be interchanged without changing the program's semantics.

3.1.1. Terms as First-Class Citizens

Given this alphabet, valid symbols can be composed. Since all data – including Prolog programs for themselves – are represented by Prolog terms, our introduction begins with the different symbols they can be built from: variables, atoms, and numbers.

Variable. A variable in Prolog is a string of letters, digits, and underscores `_`, beginning either with an uppercase letter or with an underscore. As in first-order logic, they are logic variables (cf. Section 2.4.3), and thus all the occurrences of the name of an ordinary variable stand for the same variable within one clause, i. e., the variables of the same name depict the identical entity in the domain of discourse. Only the *anonymous variable*, whose symbol is just the underscore character, is different in the way that every occurrence of `_` denotes a new, distinct variable. It is used to fill in places, where any value is allowed, not burdening the mind of the Prolog programmer with an otherwise useless name for an ordinary, fresh variable.

Like a logic program in first-order logic, a *Prolog program* consists of a finite set of definite clauses and facts. All variables are implicitly quantified, either universally in case of variables that appear in the rule's head, or existentially in case of local variables that appear only in the rule's body. Consequently, there is no need for logical symbols for explicit variable quantification in Prolog.

Atom. Unlike for first-order logic, Prolog's alphabet cannot be easily differentiate into logical and non-logical symbols. This is because a symbol's meaning is solely determined by the Prolog program it appears in. Together with the known definitions of built-in predicates provided by the ISO Prolog standard, the used Prolog system, and loaded libraries, this forms the set of allowed symbols.

Similar to first-order logic, character sequences starting with a lowercase letter are used to denote predicate symbols and function symbols. After the initial lowercase

letter, they can include digits and the underscore char `_`. In addition, any sequence of characters that is enclosed in single quotation marks `'...'` is allowed. Strings of only special characters, e. g., `==>` and `#~!`, are not required to be put in quotation marks, and are standalone valid symbols, called *graphic symbol*. A symbol, which is of one of these three classes – a string starting with a lowercase letter, enclosed in quotation marks, or consisting of only special characters –, is called an *atom*. It can be viewed as the program's smallest data item that is not divisible into parts.

Though there are no reserved symbols for the Boolean data type in Prolog, it is convention to use the atoms `true` and `false` (or alternatively `fail`), which are also built-in predicates with an arity of zero. In Prolog systems with an alphabet with full Unicode-support, the graphic symbols $\top$ and $\bot$ can be equally used as atoms. Numbers are not considered to be atoms in Prolog. Instead, they are one possible shape of terms.

Term. Syntactically, all data objects in Prolog are terms. They are inductively defined. A term is of one of the following forms:

1. A *variable* symbol of the aforementioned form, i. e., a character sequence beginning with an uppercase letter or the underscore char `_`.
2. An *atom* of the aforementioned form, e. g., `append`, `'append/3'`, or `/`.
3. A *number*. Fractional numbers use `.` as decimal point. Supported sizes and precisions depend on the Prolog system.
4. A *compound term* of the form $a(t_1,\ldots,t_n)$, where a is an atom called *functor name*, followed by $t_1,\ldots,t_n$ as the ordered set of terms as its *arguments*, which are enclosed by parentheses and separated by commas. The number of arguments is called the term's *arity*. It is greater than or equal to 1, as in case of $n = 0$ the compound term collapses to the atom a. We call a/n the *(principal) functor* of the compound term.

Atoms and numbers are summarised as *atomic terms*, as opposed to compound terms. The definition of *ground terms* applies as introduced for first-order logic in Section 2.2.

Throughout this work, we use the standard notation a/n which is based on the slash character. It is used consistently in literature to denote a predicate's functor. Only for functor names that consist only of special characters (i. e., graphic symbols), there are two different conventions. In case the functor notation should also serve as a valid Prolog term, the functor name a has to be enclosed in parentheses. Without, it would form a single atom ending with `/`. Because it is easier to grasp, we stick

to the simple notation without parentheses, i. e., `=/2` instead of `(=)/2` to denote the binary predicate with functor name `=`.

3.1.2. Rules and Facts about Predicates

A Prolog program uses rules and facts to define a predicate as a relation between its arguments.

Predicate. A *predicate* has a name p, which is an atom, and zero or more arguments. The arguments are Prolog terms, enclosed in parentheses, and separated by commas. From a syntactical point of view, a predicate is just a term – either an atom in case of zero arguments, or a compound term with the functor p/n in case of $n \geq 1$ arguments. The functor of a predicate p without arguments is defined as $p/0$.

The special meaning of predicates compared to arbitrary Prolog terms comes from their occurrences in the head of clauses, thus their meaning in the program's declarative reading.

Clause. A Prolog program consists of a finite set of definite clauses. As introduced for first-order logic, a definite clause is characterised by a single element in the positive head. A clause can be either a rule or a fact.

A *rule* in Prolog is of the following form:

```
Head :- Body.
```

`Head` is a predicate, i. e., either just p in case of a predicate with zero arguments, or $p(t_1,\ldots,t_n)$ in case of a predicate with a functor of p/n. `Body` is a Prolog goal which consists of one or more predicates. It may contain the logical conjunction, which is denoted by `,`, and the disjunction, which is denoted by `;`. The logical reading of a rule is the same as introduced for the first-order logic formula *Head* $\leftarrow$ *Body*: `Head` is true if `Body` is true.

A clause, whose body is known to be always true (or empty, respectively), is called a *fact*. In this case, it is written as follows:

```
Head.
```

The full stop `.` denotes the clause's end. The collection of all rules and facts with the same functor in the rule's head defines the corresponding predicate p/n.

In addition to facts and rules, Prolog programs can also contain directives. A *directive* is a rule with an empty `Head`, i. e., it is of the following form:

```
:- Body.
```

Directives may be placed anywhere in the Prolog source code. The goal `Body` is executed by the Prolog system when the directive is encountered, because this rule does not depend on a query due to the missing `Head`. Therefore, directives are often used to specify a program's initial query to be executed at run-time, or to provide annotations for the compiler and related tools. For instance, there are directives to define properties of a predicate for an improved compilation, and others to set program flags, or to include external libraries and source code files.

Source Code Annotations. Prolog supports two types of source code annotations, which are ignored by the Prolog interpreter:

- A *single line comment* starts with the percent sign `%` and includes everything up to the next newline character `\n`.
- A *bracketed comment* starts with the character sequence `/*` and includes everything up to the character sequence `*/`, including newline characters.

As a consequence, it is not possible to define atoms as graphic symbols, with its sequence of special characters starting with `%` or `/*`.

3.2. Working with Prolog

The main way to run Prolog programs is by Prolog's read–eval–print loop (REPL), which is called the *toplevel*. It is an interactive Prolog environment that takes a single user input, executes it, returns the result to the user and starts afresh. Terms entered at the toplevel are treated as goals, with the variables being existentially quantified. This allows to handle the goal as a query, answering the question "Is there any computed answer substitution for which the given predicate holds?" The toplevel thus outputs the computed variable bindings, or just `yes` in case of an empty answer substitution, or `no` if the goal fails.

In the work at hand, we mark code listings that illustrate the work with the toplevel by the `Toplevel` flag; queries in the running text are marked by the prefix "`?-`",

e. g., `?- halt`. We omit the trailing full stop `.` for the sake of readability, though it is actually required by the toplevel to denote the goal's end.

A Prolog source code file can be loaded in the toplevel with `consult/1`$_{\texttt{SWI}}$. Additionally, most Prolog systems expect the first argument of their toplevel executable to be a filename of the Prolog code that should be loaded.

3.2.1. Unification and Arithmetic Expressions

The toplevel computes the answer substitution by finding for each predicate in the goal corresponding clauses with a head that unifies. Therefore, the search for a most general unifier of two literals is an important feature and the foundation for the computational model of Prolog programs. The unification affects the success or failure of goals and causes the (partial) binding of free variables. We refer to [100] for a detailed description of the unification algorithm in general, and to ISO 7.3 for a description of the corresponding implementation in Prolog. Simply put, two atomic terms unify only if they are the same; two compound terms unify only if they have the same functor and their corresponding arguments unify pairwise. A free variable always unifies with a term by binding to that term.

The ISO Prolog standard defines the built-in predicate `=/2`$_{\texttt{ISO}}$ for the unification of two terms `A` and `B`. It is usually used as an infix operator, i. e., `A = B`. The additional occurs check that avoids infinite, cyclic terms in the resulting variable bindings (cf. Section 2.3) is omitted by default in all major Prolog systems for efficiency when using `=/2`$_{\texttt{ISO}}$. Implementations following the ISO Prolog standard provide the built-in predicate `unify_with_occurs_check/2`$_{\texttt{[C.6]}}$ for sound unification that detects circular variable bindings. For the operator `=/2`, Prolog implementations are free to choose the sound unification with occurs check (as implemented in `unify_with_occurs_check/2`$_{\texttt{[C.6]}}$), or use an unsound or looping unification algorithm instead, given that both work correct for all cases that are not subject to occurs check. Many Prolog systems allow to set the program flag `occurs_check(true)` to indicate that `=/2`$_{\texttt{ISO}}$ always should perform the occurs check. However, many problems can be elegantly solved in Prolog by using infinite and cyclic terms, e. g., our implementation of recursive GraphQL type systems as presented in Section 6.4.

The predicates `var/1`$_{\texttt{ISO}}$ and `nonvar/1`$_{\texttt{ISO}}$ can be used to test if a given symbol is a free variable or if the variable is already bound, respectively. In the latter case, `ground/1`$_{\texttt{ISO}}$ is true if the variable is bound to a ground term. Together it is possible to also detect partially bound variables. These three predicates are often used as preconditions in case a predicate is allowed to be used only unidirectional.

Unlike in most other programming languages, the equal sign `=` in Prolog does not cause the assignment of a term to a variable, and instead compares and unifies its two operands solely with regard to their structures. Consequently, `=/2`$_{\text{ISO}}$ also does not cause evaluation if one of the operands is an arithmetic term, i. e., `A = 1+2` simply binds the variable `A` to the term `+(1,2)`. To evaluate an arithmetic expression, Prolog provides additional predicates, which are again usually used as infix operators.

The goal `A is 1+2` causes the expression `1+2` to be evaluated and to bind the variable `A` to the result. The predicate `is/2`$_{\text{ISO}}$ is similar to the assignment operator of other programming languages, restricted to arithmetic expressions. As such, it is not commutative and requires the first operand to be either a free variable or a number, and the second operand to be a ground term. Similarly, the operators `=:=/2`$_{\text{ISO}}$, `=\=/2`$_{\text{ISO}}$, `</2`$_{\text{ISO}}$, `=</2`$_{\text{ISO}}$, `>/2`$_{\text{ISO}}$, and `>=/2`$_{\text{ISO}}$ first evaluate the two given arithmetic expressions and then compare their values.

Though it is a known concept in other programming languages, Prolog has no notation for named "don't care" variables, which are typically used to indicate positions where a term is expected that is never referenced otherwise. However, it is a common Prolog programming convention to indicate variables of that sort similar to the anonymous variable `_` by starting their symbols by the underscore character. From a semantic point of view, these longer variable symbols are of no special meaning, i. e., the variable `_var` could similarly be written as `Var` without changing the program's meaning, since all appearances of this variable symbol depict the identical entity. In all major Prolog systems, variables that follow this pattern do not throw a *singleton* warning – as otherwise, the single occurrence of a variable symbol is a strong indicator for spelling mistakes in variable names or for missing unifications. By setting SWI-Prolog's program flag `toplevel_print_anon` to `false`, variables starting with an underscore character are also not printed in the toplevel, e. g., `?- _X = 42` just returns `true`. This may be used to hide bindings in complex queries from the toplevel. Since the anonymous variable `_` has a special meaning and always introduces a new, distinct variable, its bindings are never printed in the toplevel.

3.2.2. Program Execution and Control Predicates

Prolog implements the SLD resolution as defined in Section 2.4. The SLD search tree is built depth-first, one branch at a time, using backtracking when it encounters a failure node. The clauses are applied in the order of their verbatim appearance in the source code. This behaviour can be modified by implementing a Prolog meta-interpreter, e. g., to traverse the search tree in parallel.

It is possible to cut alternative branches in the SLD search tree with the predicate `!/1`$_{\text{ISO}}$, which is called the *cut*. It discards all choice points created since entering the predicate in which it appears. If in some inner node of the SLD search tree the rule that was used to infer the current subgoal contains the cut, all following alternatives in this node are ignored, though backtracking still allows to go back to a higher node in the SLD search tree and try alternatives there. Effectively, `!/1`$_{\text{ISO}}$ is a goal which always succeeds, but cannot be backtracked past. It is often used for efficiency reasons or for shorter source code, since preconditions of alternative rules with the same head have to be stated only in one of the rule's body.

Prolog relies on the closed-world assumption to handle negation – its SLD resolution can tell if a goal is false by trying to prove it. If this attempt fails, it concludes that the proposition is false. A negated goal can be stated with the predicate `\+/1`[C.2], which is supposed to be a mnemonic for "not provable" with the backslash `\` as Prolog's symbol for negation, and the plus `+` for "provable". The implementation of the `\+/1` is based on the cut, as given in Appendix C.2.

3.2.3. Properties of Predicates and Programs

In this section, we introduce notions for properties of Prolog predicates and programs. They are used throughout this work to describe predicates in source code annotations [136, Sec. 4.1 and Sec. 5] and might be processed by automatic source code documentation infrastructure [49, 131], as well as by directives for the compiler, e. g., for verification and static type analysis [75] and improved efficiency [123].

Termination. The negation with `\+/1`[C.2] illustrates that there are two kinds of termination in Prolog: *universal termination*, and *existential termination*. By backtracking, all possible answer substitutions are computed for a positive goal `G`. If there are infinitely many solutions, the query `?- G, false` never terminates – it does not *terminate universally*. In contrast, the negated goal `\+(G)` is known to be false if there is just one solution for the goal `G`. Consequently, the query `?- \+(G), false` terminates in case `G` *terminates existentially*, giving at most one solution. Whether a predicate terminates existentially or terminates universally, is undecidable in general.

In its declarative reading, the computed answer of a query that ends with `false/0`$_{\text{ISO}}$ necessarily will always be `false`. It is, however, a common construct in Prolog programs to force backtracking over all solutions, called *failure-driven loop*.

Determinism. This problem arises from Prolog's design to express relations instead of functions. In functional programming languages like Haskell, a function returns at most one result, while in Prolog a relation can describe multiple entities which are all returned by backtracking. The predicate's property, how often it is intended to succeed, is called *determinism*, and is one of the following:

- A predicate is called *semi-deterministic* (abbreviated in source code annotations as `semidet`), if it is expected to fail or to succeed with only a single answer that is always the same for the same input. For instance, the predicates for type tests `var/1`$_{\texttt{ISO}}$ and `acyclic_term/1`$_{\texttt{ISO}}$ are semi-deterministic.
- A predicate is called *deterministic* (`det`), if it succeeds exactly once. For instance, the predicate `is/2`$_{\texttt{ISO}}$ is deterministic for valid arithmetic expressions.
- A predicate is called *multi-deterministic* (`multi`), if it succeeds at least once. It is often used for generators.
- If otherwise the predicate can succeed arbitrarily often (including never), it is called *non-deterministic* (`nondet`).

These four determinism values originate from the declarative logic programming language *Mercury* [112]. In literature, the first two are often simply referred to as *deterministic*, while the latter two are subsumed under the phrase *non-deterministic*. However, all four kinds are common in Prolog source code annotations to describe the backtracking properties of a Prolog predicate in more detail.

Note that our definition of nondeterminism differs from those of algorithms: though a predicate is non-deterministic, for the same inputs it usually computes the same answer substitutions in the same order.

Built-in and User-defined Predicates. The query we used as an example for non-termination contains the goal `false/0`$_{\texttt{ISO}}$. Together with about a hundred other predicates it is defined in the ISO Prolog standard (Iso 8); additional predicates that can be directly queried are usually provided by the used Prolog system. They form the set of *built-in predicates*. On the other hand, own predicates that are implemented by facts and rules are called *user-defined*.

Some built-in predicates are shipped in a library, e. g., `append/3`$_{[3.4]}$ is part of SWI-Prolog's *library(lists)*. It can be loaded by calling `?- use_module(library(lists))`. In addition, SWI-Prolog supports *autoloading*: when a predicate is found missing at run-time, its implementing library is searched and the predicate is imported lazily using the predicate `use_module/2`$_{\texttt{SWI}}$.

Variable Instantiation and Logical Pureness. In theory, Prolog predicates just describe relations. In practical applications though, not all problems can be described this way. For instance, as mentioned in Section 3.2.1, the built-in predicate `is/2`$_{\texttt{ISO}}$ requires the second argument to be ground. Although for the goal `A is 1+2` the variable `A` is bound to the number 3, `is/2`$_{\texttt{ISO}}$ cannot be used the other way round to generate all terms that evaluate to 3.

As can be seen, there are predicates with requirements regarding the content of their arguments. The set of constraints per argument is called the predicate's *(instantiation) mode*. It is denoted by a tuple of symbols that correspond to each of the predicate's argument:

- `+` The argument has to be (partially) bound. This likely describes one of the predicate's input arguments.
- `++` The argument has to be ground, thus a stricter mode than `+`.
- `-` The argument should be a free variable. This likely describes one of the predicate's output arguments.
- `--` The argument has to be a free variable. This is a stricter mode than `-`. It acknowledges the common use case that a predicate's output is tested to be a known value. In this case, even for a mode `-` the argument would already be bound.
- `?` There are no restrictions to the argument. This likely describes an argument that is used alternatively as the predicate's input or output, or serves as both (e. g., in case of a partially bound variable).
- `@` The argument will not further be instantiated. Typically, this argument is used only for type tests, e. g., via `var/1`$_{\texttt{ISO}}$ or `acyclic_term/1`$_{\texttt{ISO}}$.
- `:` The argument will be meta-interpreted in some way.

For instance, the built-in predicate `is/2`$_{\texttt{ISO}}$ has the mode `(?,+)`. The mode is often specified in the predicate's description, for instance as part of source code annotations. There, the symbols are used as a prefix for a variable name that describes the predicate's argument, e. g., in `is(?Number,+Expression)`. This detailed notation of the mode is called the predicate's *signature*.

A single predicate might have multiple modes. For instance, SWI-Prolog's implementation of `char_code/2`$_{\texttt{SWI}}$ supports the modes `(+,-)` and `(-,+)`, i. e., at least the character or the code has to be bound to get the other. Our implementation in Appendix C.5 on the other hand has no such restriction and follows the mode `(?,?)`. It can thus be used to backtrack over all pairs by calling `?- char_code(Char,Code)`

with free variables `Char` and `Code`. In this work's listings that define Prolog predicates, we specify their instantiation modes in source code annotations placed at the beginning.

Dependence on the type of an argument is a sign for a *(logical) impure* predicate. The instantiation modes `--` and `-` differ only for predicates which are not *steadfast*. This is the case for a predicate g, which has an argument `V`, that succeeds for the query `?- g(V), V = t`, but fails for `?- V = t, g(V)`, with t being a term but a free variable.

3.3. Data Structures

In contrast to other programming languages, Prolog has a limited number of built-in data types and data structures. Besides variables, atoms, and the primitive data type of numbers, it only allows the construction of compound terms to model complex data. To test the type of a symbol, the ISO Prolog standard defines several deterministic predicates. With `var/1`$_{\texttt{ISO}}$ and `nonvar/1`$_{\texttt{ISO}}$, a given symbol is tested to be a free variable or if the variable is already bound. In the latter case, `atomic/1`$_{\texttt{ISO}}$ is true if the term is not bound to a compound term, i. e., it is an atom, string, integer, float, or number in general – which can be tested via the built-in predicates `atom/1`$_{\texttt{ISO}}$, and `string/1`$_{\texttt{ISO}}$, `integer/1`$_{\texttt{ISO}}$, `float/1`$_{\texttt{ISO}}$, and `number/1`$_{\texttt{ISO}}$, respectively. The predicate `compound/1`$_{\texttt{ISO}}$ on the other hand is true if the given term is of no primitive type and, instead, is bound to a compound term. Together with atoms, they make up the class of *callables*, which can be tested with the predicate `callable/1`$_{\texttt{ISO}}$. This is a type test typically performed for arguments to meta-predicates like the family of `call/`$n_{\texttt{ISO}}$. However, it only tests the general structure of the given bound term's surface. Though terms like `(true, integer, 3)` follow the structural requirements to be treated as a callable, the goal `?- true, integer, 3` cannot be executed and raises an error because of the unknown predicate `integer/0`, and a type error for calling the integer value `3`.

Complex data in Prolog can be built as compound terms. With its inductive definition, they represent hierarchical data structures, and thus naturally correspond to trees, with atomic terms and variables as their leaves. In this section, we introduce compound term structures that are commonly used in Prolog applications and throughout the work at hand.

3.3.1. Lists

As introduced for first-order logic in Section 2.5, lists are simply trees where each node has a single child and holds the list element's value. Instead of *nil*, Prolog uses the atom `[]` to denote the empty list. The atom `.` is used as the *constructs* function in the ISO Prolog standard. This way, the list which contains only the atoms `a` and `b` is represented by the term `.(a,.(b,[]))`.

Instead of this verbose notation, the ISO Prolog standard also allows to write a list as a comma-separated sequence of terms, enclosed by square brackets. The aforementioned list thus is the same as the term `[a,b]`.

Terms with the *constructs* function `./2`$_{\texttt{ISO}}$ can also be written using the square bracket notation: the term `[H|T]` is the same as `.(H,T)`. This notation is frequently used to split a list into its *head* `H` and the *tail* `T`, or vice versa to construct a list by specifying its first element and the remaining list. Both variants can be combined in any way, e. g., `[a,b|[]]` represents the same list of two elements `a` and `b`.

The predicate `is_list/1`$_{\texttt{SWI}}$ can be used to check if the given variable is bound to a list, i. e., it is either the empty list `[]`, or the term `[_|T]`, with `T` being a list again. Though not part of the ISO Prolog standard, this predicate is part of all major Prolog systems, and is autoloaded in SWI-Prolog's *library(lists)* [136, Sec. 4.29].

3.3.2. Pairs

A pair can be constructed as a compound term with a functor *p*/2. For instance, the term `pair(a,b)` denotes the pair of the atoms `a` and `b`. By defining and using `pair/2` as an infix operator, the term can be equally written as `a pair b`. With graphic symbols, i. e., atoms built from only special characters, this results in a short and descriptive notation, e. g., `a-b`. Typical infix operators to denote pairs are `-/2`$_{\texttt{ISO}}$, `:/2`$_{\texttt{ISO}}$, `//2`$_{\texttt{ISO}}$, or `=/2`$_{\texttt{ISO}}$. The latter does not conflict with the ISO Prolog standard predicate for unification, as it is used in the term as a function symbol instead of its meaning as a predicate. Sometimes, a pair is also represented by the compound term `Key(Value)` (e. g., in SWI-Prolog's autoloaded *library(pairs)*). Though this notation does not allow to backtrack over all keys in case of unification, as the functor name of a compound term is required to be an atom in Prolog and thus `Key` cannot be left as a free variable.

3.3.3. Difference Lists

A typical example to use a pair with functor `-/2`$_{\text{ISO}}$ is the notation of a *difference list*. It is a pair of a normal list, whose tail is a logic variable `T`, and `T`. For instance, `[a,b|T]-T` denotes the pair of a list `[a,b|T]` and the variable `T`. The list contains the atoms `a` and `b` as its first elements.

A difference list is called *open (difference) list* (or alternatively *partial list*), if `T` is a free variable or again an open list. In contrast, if `T` is bound to ground term, the difference list is called *closed (difference) list*. In this notation, the list which contains only the atoms `a` and `b` is represented by the closed difference list `[a,b]-[]`.

Difference lists are a means to address the performance issues coming from Prolog's lack of lists with random access, as it is provided by arrays in other programming languages. Instead of first having to consecutively traverse through the whole list to add a new element `c` at the very end, `T` can be simply partially bound to the term `[c|T2]`, resulting in the open list `[a,b,c|T2]-[c|T2]`, with `T2` being a free variable and `[c|T2]` an open list again, which thus can be processed alike.

The previous example unveils the origin of the phrase "difference list": the original list `[a,b]` is the difference (happily using the atom `-` as its infix operator) of the lists `[a,b|T]` and `T`. Practical applications of difference lists include problems where the resulting list gets lazily evaluated, e. g., when reading in files in chunks (cf. Section 6.1), or Prolog's definite clause grammars (cf. Section 6.2).

3.3.4. Strings

Though there is a dedicated syntax to denote strings by enclosing them in the double quote character `"`, Prolog has no corresponding primitive data type. Instead, a string can be thought of as a sequence of characters, and thus be represented by a list. Treating strings this way has a long tradition in Prolog. It has several advantages, e. g., it allows to define strings with holes by using free variables as list elements. All predicates which originally are defined for lists work on strings, too. However, there is a recent debate in the Prolog community on how to handle strings [87, 129]. SWI-Prolog 7 introduced a new primitive data type for strings: text enclosed in double quote characters is read as an object that lives on the global stack. With strings regarded as lists, there are also different notions. Some systems represent each character as a one-character atom, others use the character's code as integer instead. In the latter case, `"abc"` is the same as the term `[97,98,99]`.

We do not elaborate on the discussion of the preferred format, as from a syntactical point of view, the original notation `"abc"` in the Prolog source code is the same for all variants. In addition, all Prolog systems following the ISO Prolog standard allow to determine how the string `"abc"` is read in by Prolog via the flag `double_quotes`. It sets the internal representation of strings as one of:

- `codes`: `[97,98,99]`
  The string represents a list, with each element being the integer code of the corresponding character as returned by `char_code/2`[C.5].
- `chars`: `[a,b,c]`
  The string represents a list of one-character atoms. This approach is convenient and makes debugging easy. In addition, it hides encoding problems from the programmer as it does not rely on a conversion from the character to its code via `char_code/2`.
- `atom`: `abc`
  The string represents the corresponding atom, possibly enclosed by single quotation marks. This option is typically used only in cases where identity comparison is the main operation and where strings are not processed further.
- `string`: `"abc"`
  The string represents a primitive data type available in SWI-Prolog 7 and higher. SWI-Prolog's predicate `string/1`$_{\texttt{SWI}}$ can be used to test if a given variable is bound to this atomic data type.

In strings, the backslash `\` serves as an escape character, e. g., in `"\"\\"`, which denotes `["\]`. In addition to strings which are enclosed by the double quote character `"`, the ISO Prolog standard similarly defines strings enclosed by the back quote character `` ` ``, though its meaning is left open. Some Prolog systems use these two kinds to read in the enclosed character sequences in different internal representations.

3.3.5. Dicts

A similar idea as building lists from compound terms can be used to mimic a data type for named key-value associations, that is often available in other programming languages. The pairs of key and value can be modelled in the form of a compound term with two arguments, which are then put in a list.

As an example, instead of using a predicate called `person/2` with an argument for each property, the person's data can be modelled as follows:

```prolog
person(1, [firstName='Alice', birth=1986]).
```

This comes with a great flexibility, as the number of keys is not limited by the predicate's functor. Additional key-value pairs can be easily amended without having to change all predicate calls for the adjusted arity. In addition, the `person/`n facts are not required to all have the same, fixed arity of n, which is useful for optional values, e. g., in case of an unknown birth year.

Using the list representation, a single value can be retrieved just by unification, e. g., using `memberchk/2`[C.8] to get the person's first name:

```prolog
first(Id, Name) :-
  person(Id, KV),
  memberchk(firstName=Name, KV).
```

However, representing key-value associations as a list in Prolog has several disadvantages. Firstly, this notation does not ensure the uniqueness of contained keys. And secondly, it does not provide a short notation to directly access a child node. As a result, accessing data comes with the linear worst-time complexity known from normal lists due to their internal representation as compound terms.

SWI-Prolog 7 introduced *dicts* as a new data type for named key-value associations [129]. Their syntax resembles the one of the JavaScript Object Notation (JSON). Because JSON is a popular data exchange format for web applications, this new syntax of dicts should be easier to use for developers of other programming languages than Prolog's traditional key-value association lists, and simplifies the development of web services with SWI-Prolog.

SWI-Prolog's dicts are of the following form:

```swi-prolog
Tag{ Key1: Value1, Key2: Value2, ... }
```

Compared to JSON, every dict has a leading tag `Tag`, which can be used to name the type of the dict. The tag is either an atom or a variable. Since the variable does not need to be bound, we can use the anonymous variable `_` as the dict's tag, resulting in an *anonymous dict* `_{...}`. The curly brackets can contain an arbitrary number of key-value pairs. Each key is an atom and has to be unique within the dict. The associated value can be any valid term. In particular, it is possible to create nested dicts.

For instance, the person's data can be similarly represented with the help of an anonymous dict instead of the association list we used before:

```
person(1, _{ firstName: 'Alice', birth: 1986 }).                SWI-Prolog
```

Unlike for lists, the value in a dict can be directly accessed by its key. SWI-Prolog provides two methods:

- `Dict.Key` retrieves the value of the key `Key` in a dict `Dict`. This is called the *dot notation* for dicts. `Dict` has to be bound to a dict, `Key` is allowed to be a free variable, i. e., the mode is `.(+Dict,?Key)`.

- `Dict.get(Key)` has the same effect as `Dict.Key`, but silently fails instead of throwing an error if the specified key `Key` does not appear in the dict `Dict`. It can therefore be also used as a precondition when expecting a particular key. Its mode is the same as for `./2`, i. e., `+Dict.get(?Key)`.

Both notations rely on the infix operator `./2`[C.13], which was introduced in SWI-Prolog 7 together with dicts. Internally, goals that contain the infix operator `./2`[C.13] are compiled to calls of the predicate `./3`[C.14].

Dicts can be unified following the standard symmetric Prolog unification rules, although the unification will fail if both dicts do not contain the same set of keys. For partial unification, SWI-Prolog provides the two infix operator `:</2`[C.15] and `>:</2`[C.16]. `Select :< From` is true if the association list corresponding to the dict `Select` is a subset of those of the dict `From`. `Dict1 >:< Dict2` is true if the values of all keys that appear both in `Dict1` and `Dict2` can be unified. In addition, both operators `:</2`[C.15] and `>:</2`[C.16] unify the tags of their operands. Listing 3.1 gives various examples of using these operators.

Because of using the infix operator `./2` for functions on dicts, SWI-Prolog version 7 breaks with the classical notation of lists as defined in the ISO Prolog standard (cf. Section 3.3.1). SWI-Prolog instead relies on the operator `[|]/2`$_{\text{SWI}}$ for list construction. This breaking change of SWI-Prolog version 7 has been widely discussed in the logic programming community and appears not be considered for adoption either in the standard or other implementations of Prolog.

Besides the proprietary data structures of dicts, there are alternative, native Prolog term representations to express nested data. Among others, Seipel et al. introduced the *field notation* [104], which is based on association lists and triples of the form `Type:Attributes:Children`. It integrates a declarative query mechanism called FnQuery [107], and is, e. g., used to represent and query XML documents in Prolog.

Listing 3.1: Example queries for the unification of two dicts.

Toplevel SWI-Prolog

```
?- p{ a: 1, b: 2 } = P{ a: A, b: B }.
P = p, A = 1, B = 2 .

?- p{ a: 1, b: 2 } = P{ a: A }.
false .

?- P{ a: A } :< p{ a: 1, b: 2 }.
P = p, A = 1 .

?- p{ a: 1, b: 2 } :< P{ a: A }.
false .

?- p{ a: 1, b: 2 } >:< P{ a: A, c: 3 }.
P = p, A = 1 .
```

3.4. Prolog Example: `append/3`

The definition of the predicate *append* from Section 2.5 can now be easily rewritten as a Prolog program. Listing 3.2 presents three possible implementations, where the first is in lines 1–5. It uses two clauses, with line 3 corresponding to the fact of formula F_1, and the recursive rule of F_2 implemented in lines 4–5. The mode of `append/3` is `(?,?,?)` – as retraced step-by-step in Sections 2.5.1 and 2.5.2, the predicate can be called differently, making it suitable for both concatenating and splitting lists.

The second approach depicted in lines 7–10 of Listing 3.2 uses the combined formula F' instead. The logical disjunction is made explicit by the built-in predicate `;/2`$_{\text{ISO}}$ and moved to the rule's body. Though the declarative reading is the same for both variants, the first implementation of lines 1–5 should be preferred. It allows the compiler to statically analyse the clauses of `append/3` and index them by their arguments, which results in faster execution times [17, 123]. Today's Prolog systems all support indexing of the predicate's first argument (which is sufficient for our implementation of `append/3`), and most also of multiple arguments, e. g., SWI-Prolog [136, Sec. 2.18].

The third alternative implementation in line 13 of Listing 3.2 is targeted to difference lists (cf. Section 3.3.3) instead. The underlying concept of this implementation is as follows: a list `C` can be built from two lists `A` and `B`, if `A` is `C` without ("minus") `B`. Since `C` is a difference list, it is of the form `Z-T`. `B` is also a difference list, and must have the same tail as `C`, so it can be represented by `Y-T`. Then, `A` is the list `Z` without

Listing 3.2: Three alternative implementations of the predicate `append/3`, based on the formulas which we introduced in Section 2.5, and difference lists.

PROLOG

```
1  %% append(?List1, ?List2, ?List1_then_List2)
2  % implementation based on F1 and F2
3  append([], Y, Y).
4  append([E|X], Y, [E|Z]) :-
5    append(X, Y, Z).
6
7  % alternative implementation, based on F'
8  append(X, Y, Z) :-
9    ( X = [], Y = Z
10   ; X = [E|L], Y = [E|M], append(L, Y, Z) ).
11
12 % alternative implementation with difference lists
13 append_difflists(Z-Y, Y-T, Z-T).
```

Listing 3.3: Example queries for `append/3` and `append_difflists/3`.

TOPLEVEL

①
```
?- append(X,Y,[a,b]).
X = [], Y = [a, b] ;
X = [a], Y = [b] ;
X = [a, b], Y = [] ;
false .
```

②
```
?- append(_,[a|_],L).
L = [a|_] ;
L = [_, a|_] ;
L = [_, _, a|_] ;
% ... and more solutions
```

③
```
?- append_difflists([a|T1]-T1, [b|T2]-T2, L).
T1 = [b|T2], L = [a, b|T2]-T2 .
```

("minus") `Y`. The fact in line 13 simply puts the described unifications `C = Z-T`, `B = Y-T`, and `A = Z-Y` in the clause's head.

Since the implementation with difference lists uses only a single unification, it is of complexity $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$ in the traditional `append/3` implementation, with n as the list length of the first argument. However, unlike `append/3`, the version relying on difference lists cannot be used to split a list. Nothing in Prolog enforces the constraint that the tail (i. e., the second argument in the `-/2`$_{\text{ISO}}$ pair) of a difference list is a sublist of the list (i. e., the first argument). Therefore, in our implementation of `append_difflists/3` in Listing 3.2, for goals with free variables in the first two arguments, `Y` will bind to something which is neither a sublist of `Z` nor a superlist of `T`.

Listing 3.3 gives three examples of using these predicates in the toplevel for various questions. In ①, we show three computed answer substitutions for two lists `X` and `Y`, which are requested to form together the list `[a,b]`. The results and their

order are the same as we got in Section 2.5.1. The second example in (2) illustrates that `append/3` can be used to implement the predicate `member/2`. Since `?- member(?Elem,?List)` is true if `Elem` appears in the list `List`, we can alternatively search via `append/3` for a partial list in `List` that starts with `Elem`. The goal of (2) terminates only existentially but not universally, as in each step n the open list that has a set at the n-th position is returned. Finally, the example (3) addresses the implementation with difference lists. In the process of concatenation, `T` has to be the list that starts with `b`, so `T1` gets partially bound.

Besides the ternary `append/3`, many Prolog systems ship with the predicate `append/2`, which concatenates a list of lists. Unlike the more general `flatten/2`, which we describe in Appendix C.12, it handles only well-formed two-dimensional lists.

3.5. Reflection and Code Listings

It is worth noting that the recursive rule in the `append/3` Prolog program of Section 3.4 (lines 4–5 of Listing 3.2) itself is again a valid Prolog term. Given that $\texttt{:-/2}_{\texttt{ISO}}$ is an infix operator, the rule's head is its first operand, and the rule's body the second. In fact, all program information encoded in a Prolog source can be completely represented just by Prolog data structures. In its declarative reading, clauses again are first-order logic predicates, or terms from a syntactical point of view. This property of a programming language is called *homoiconicity*.

Some programming languages (e. g., Java, Python) offer the ability to examine, introspect, and modify their own structure and behaviour. This capability is called *reflection*. Since Prolog is a homoiconic language, Prolog programs are valid Prolog terms. They can be analysed and processed like any other Prolog term. This allows reflection and program transformations – from a syntactical point of view, there is no distinction between code and data. All clauses are treated as data, with means for adding, removal, and modification, as we introduce in Section 3.7. Consequently, the set of all clauses of a loaded Prolog program is also called *database*.

However, it is not possible to select all rules of a predicate p/n with a head of $p(\texttt{V}_1,\ldots,\texttt{V}_n)$ by simply calling `?-` $p(\texttt{V}_1,\ldots,\texttt{V}_n)$ `:- Body.` in the toplevel, because though the ISO Prolog standard defines `:-/2` as an infix operator, the standard does not define it as a Prolog predicate. Instead, facts and rules are of special meaning – their head is unified during SLD resolution and is often indexed for better performance. Therefore, Prolog offers the predicate $\texttt{clause/2}_{\texttt{ISO}}$. The goal `clause(:Head,?Body)` backtracks over all clauses that unify with `Head`; for facts, `Body` is bound to $\texttt{true/0}_{\texttt{ISO}}$. This way, it is possible to find meta data related to a

goal in the database. As a result, `clause/2`$_{\text{ISO}}$ allows to create metacircular interpreters and write evaluators that use non-standard search orders [88]. To address the initial question, this allows to explicitly define the predicate `:-/2` by the rule `:-(H,B) :- clause(H,B).` – though we strongly advise against, as it may break third-party modules. Given this predicate definition, querying clauses in the form of the goal `?- (Head :- Body)` is possible.

Prolog provides the predicates `listing/`$\{0,1,2\}_{\text{SWI}}$ in addition to the standard `clause/2`$_{\text{ISO}}$, mainly for debugging purposes. They allow to print the source code of a complete module (arity 0) or of a given predicate (arity 1) in a human-readable format, possibly with options set for the output (arity 2). They are part of *library(listing)*, which is autoloaded in SWI-Prolog. The produced text-based listing includes relevant declarations, like `dynamic/1`. It cannot be used for reflection, because the source code information is printed to the standard output stream instead of binding to a variable.

3.6. Term Inspection and Higher-Order Predicates

While reflection is used to examine the rules of a Prolog program, there are also dedicated predicates for *term inspection*. They are an alternative for unification that allow to decompose and analyse terms:

- `functor/3`$_{\text{ISO}}$ can be used to compose and decompose compound terms. `functor(?Term,?Name,?Arity)` is true when `Term` is a term with `Name`/`Arity` as its functor.

- `arg/3`$_{\text{ISO}}$ allows accessing a single argument of a given compound term. `arg(?N,+Term,?Value)` is true when `Value` is the `N`-th argument of the term `Term`.

- The `=../2`$_{\text{ISO}}$ predicate is called *univ* and is typically used as an infix operator. `?Term =.. [?Name|?Args]` is true when `Term` is a term with a functor name of `Name` and `Args` as a (possibly empty) list of its arguments.

These predicates cannot be defined by a finite set of clauses as originally required by first-order logic, and can therefore be considered higher-order predicates. Besides terms, they can be equally used to create predicates. This is in particular useful when working with *higher-order predicates* or *meta-predicates*, i. e., Prolog predicates that take predicates as their arguments.

Meta-Predicates `call/`$n_{\texttt{ISO}}$. A popular example for meta-predicates is the family of `call/`$n_{\texttt{ISO}}$ predicates. With the presented predicates for term inspection, it is possible to create goals that are not known at compile-time, and call them at run-time. The built-in predicate `call/`$n_{\texttt{ISO}}$ takes a goal as its first argument, appends all following arguments to the goal's argument list and calls the result. The ISO Prolog standard requires the meta-predicates `call/`$n_{\texttt{ISO}}$ defined for arities of $1 \leq n \leq 8$. Higher arities are supported by many Prolog systems as well, but usually handled by the compiler, i. e., they are not apparent for Prolog's reflection and code listing capabilities. The meta-predicate `?- call(Goal)` acts as if the variable `Goal` was written plain as a goal in the rule's body, except for restricting the scope of possibly contained cuts. A cut inside `?- call(Goal)` only affects choice points created by `Goal`, which becomes clear by the predicate's underlying implementation as a separate clause:

```prolog
%% call(:Goal)
:- meta_predicate call(0).
call(Goal) :- Goal.
```
PROLOG

The definitions of the meta-predicates `call/`$n_{\texttt{ISO}}$ with $n \geq 2$ rely on the *univ* operator `=../2` for term inspection and the dynamic construction of the applied goals at run-time. An exemplary implementation of `call/2`$_{\texttt{ISO}}$ is given in Appendix C.1.

Meta-Argument Specifiers. The directive `meta_predicate/1`$_{\texttt{SWI}}$, which we used in the aforementioned implementation of `call/1`$_{\texttt{ISO}}$, declares a meta-predicate and its arguments. This information is used by the compiler for cross-referencing predicates, to denote module-sensitive arguments, as well as for debugging purposes and syntax highlighting. The directive `:- meta_predicate` *`Head`*`(Spec`$_1$`, ..., Spec`$_n$`)` declares the meta-predicate functor *`Head`*`/`n and provides a *meta-argument specifier* `Spec`$_i$ for each i-th argument of the predicate. Typical meta-argument specifiers are:

- `+` The argument is (partially) bound on all calls of the meta-predicate.
- `-` The argument is a free variable on all calls of the meta-predicate.
- `?` The argument is either a variable or (partially) bound.
- `N` An integer value `0`, ..., `9` denotes an argument that is a term referring to a predicate with `N` more arguments.
- `//` The argument denotes a goal of a definite clause grammar, which we introduce in Section 6.2. Typically, this argument is a nonterminal defined in a grammar rule's left-hand side.

At positions with a meta-argument specifier of `N` or `//`, no free variable is allowed. Consequently, it is not possible to answer the question "Is there *some* predicate of functor `P/(`n`+N)` that can be called with the given arguments?", which aligns with first-order logic's restriction to not support propositions about predicates.

As seen before, the unary meta-predicate `call/1`$_{\texttt{ISO}}$ has a single meta-argument specifier `0`, as its first and only argument is a predicate of functor `P/`n. The predicate `call/2`$_{\texttt{ISO}}$, on the other hand, is declared as `:- meta_predicate call(1,?)` because its first argument is a term of functor `P/`n that refers to the predicate `P/(`n`+1)` – since the second argument of `call/2` is appended as an argument to the term `P/`n before it is called. Analogous, each predicate of the `call/`n family has the number $n-1$ as its first meta-argument specifier which is followed by $n-1$ arguments with specifier `?`.

Toplevel Internals. As an example for the usefulness of the family of `call/`$n_{\texttt{ISO}}$ predicates, we consider Prolog's toplevel. It is usually implemented as a failure-driven loop and uses the meta-predicate `call/1`$_{\texttt{ISO}}$ together with the predicate `read/1`$_{\texttt{ISO}}$, which reads a Prolog term from the current input stream and unifies it with the variable given as its argument. The toplevel essentially performs the goal `?- read(G), call(G), write(G), false` to first read a term `G` from the input stream, call it, and write the possibly (partially) bound term back to the user. With `false/0`$_{\texttt{ISO}}$, these steps are backtracked to obtain all results for `G`, and repeatedly ask the user for further goals. The approach with a failure-driven loop though removes all globally set variables and constraints when backtracking over `call(G)`. We therefore recommend to implement the toplevel as a recursive loop instead. Our suggestion has been adapted by SWI-Prolog of version 7.3.28, which allows to set the used mode by the program flag `toplevel_mode` [136, Sec. 2.12].

3.7. Dynamic Predicates

The means for reflection we presented in the previous section are focussed on just examining the clauses of an already loaded Prolog program. In addition to this, the facts and rules can also be modified at run-time. This way, data can be stored that persists the program's lifecycle. The ISO Prolog standard defines multiple predicates to work with the Prolog database:

– The built-in predicate `asserta/1`$_{\texttt{ISO}}$ and `assertz/1`$_{\texttt{ISO}}$ dynamically asserts a given clause as first (suffix `a`) or last clause (`z`) of the predicate it defines.

Listing 3.4: Implementation of global variables using a dynamic predicate `'$val'/2` for data storage.

```
1  % declare predicate '$val'/2 for data storage as dynamic
2  :- dynamic('$val'/2).
3
4  %% setval(+Name, +Value)
5  setval(Name, Value) :-
6    retractall('$val'(Name,_)),
7    assert('$val'(Name, Value)).
8
9  %% getval(?Name, ?Value)
10 getval(Name, Value) :-
11   '$val'(Name, Value).
```

PROLOG

- Given a term `T`, the goal `?- retract(T)` unifies with the first matching clause head in the database and removes this rule or fact. It supports backtracking and fails in case there is no unifying clause head.
- The predicate `retractall/1`$_{\text{ISO}}$ deletes all clauses that unify with a given term – possibly none, thus it is always succeeding.

Compared to classical database systems, there is no update command. However, with the help of `assert/1`$_{\text{ISO}}$ and `retractall/1`$_{\text{ISO}}$, it is possible to define a predicate `setval/2` to replace one clause by another, which is for instance typically done to mimic global variables. Listing 3.4 provides an exemplary implementation. The predicate `getval/2` retrieves an already set value. Both predicates are based on the data storage in facts of `'$val'/2`, which is a dynamic predicate. Its name follows Prolog's convention to denote internal predicates by functor names starting with the dollar symbol. Following the ISO Prolog standard, predicates that are modified dynamically at run-time need to be declared using the `dynamic/1`$_{\text{ISO}}$ directive (Listing 3.4, l. 2). In the implementation of `setval/2`, the call of `retractall/1`$_{\text{ISO}}$ (l. 6) succeeds even if there is not yet a clause `'$val'/2` in the database, thus no data initialisation is required.

3.8. Modules

For programming in the large and to improve the reusability of Prolog code, it is required and advised to split codebases into smaller logical components. *Part II* of the ISO Prolog standard [56] defines a module system for Prolog. Because of Prolog's long history, module systems have received considerable attention before and after *Part II* of the ISO Prolog standard was published in 2000, e. g., in [19, 45, 74].

Though today most Prolog systems implement some kind of module system, they are often differently focussed and incompatible. This mainly arose from the system's long lifetime, as well as conflicting requirements to their module systems. For instance, GNU Prolog [32] initially did not implement any module system and opted for the implementation of a cleaner yet proprietary alternative mechanism instead, which is called *contextual logic programming* [3]. The module system of Ciao [50] on the other hand extends the ISO Prolog standard to allow separate compilation and creation of standalone executables based on modules [16] and allows to selectively avoid the loading of predicates defined in the ISO Prolog standard. The portability of the module systems provided by Ciao, SICStus Prolog [18], YAP [24], and SWI-Prolog is compared in [133].

In addition to the systems' long histories, there are many reservations regarding the module system defined in the ISO Prolog standard, so it is not implemented by any of the major Prolog systems. Though they are consequently incompatible, all provided module systems provide means to structure Prolog source code into distinct files for separation, modularisation, and to avoid name conflicts of predicates. The *Prolog Commons*[10] working group was founded in 2009 in order to start to address some of the compatibility problems and issues with the current state of *Part II* of the ISO Prolog standard [128].

In SWI-Prolog, a module is treated as a collection of predicates and operators which defines a public interface. Its syntax is derived from the Quintus Prolog module system. This module system was the starting point for a number of Prolog systems, such as SICStus Prolog, Ciao and YAP, thus most of them provide compatibility layers or guidelines for migrating existing modules from one system to the other.

By default, there are two modules:

- The `system` module holds all built-in predicates and operators, including those from libraries shipped with SWI-Prolog like *library(lists)*.
- The `user` module contains all user-defined predicates and operators and is initially empty. It forms the working space of the user, i. e., predicates and operators which are not defined in some other modules are part of this `user` module. It automatically imports the `system` module, thus making all built-in predicates available.

A new module can be defined using the $\texttt{module/}\{2,3\}_{\texttt{SWI}}$ directive at the very beginning of the Prolog source code file. The directive declares the module's name as an atom in its first argument and lists the public (i. e., externally visible) predicates

[10] Prolog Commons, https://prolog-commons.org.

and operators as its second. The third, optional argument can be used to declare the system's dialect and is supported for compatibility with the *Prolog Commons* initiative. All clauses following the `module/`$\{2,3\}_{\texttt{SWI}}$ directive are loaded into this module. Since the `user` module (and consequently the `system` module) is always imported, all user-defined and built-in predicates can be used without explicitly importing them.

Importing the predicates from another module is achieved using the directive `use_module/`$\{1,2\}_{\texttt{SWI}}$. It takes a filename as its first argument. This is why modules should be defined in a file of the same name. The optional second argument takes a list of predicates and operators which should be imported. A major advantage is that internal predicates of modules, as well as the module's public predicates which are not imported, cannot cause name conflicts with the predicates of other modules, including the `user` module. If two different modules define a predicate with the same functor, they can be addressed by explicitly stating the module via the infix operator `:/2`$_{\texttt{SWI}}$, e. g., in `?- lists:append([a],[b],X)`.

4

Domain-Specific Languages

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

— Martin Fowler[11]

If one asks developers for their most favourite programming language, the answers will be of large variety. Front-end developers might prefer Java or Swift; web developers choose JavaScript, HTML and CSS; and Python and Prolog could be the languages of choice for researchers in the field of artificial intelligence and machine learning. In each of these areas, there are only a handful of predominant programming languages – apparently, they are already somehow specific to their application domain. Though this is often not the result of the language's expressiveness, but a community standard.

Nevertheless, modern software does not reside in a closed world. Every system needs to interact with its users and other software components, with the help of standardised interfaces or data exchange formats. Therefore, developers are not only faced with their single favourite computer language, but also have to integrate additional languages and data formats.

As an example, we consider the relational database system *MySQL*.[12] It uses the *structured query language* (SQL), a domain-specific language for data query, data manipulation, data definition, and data control over relational tables. For instance, the query `SELECT * FROM users` returns all rows and columns from the table `users`. The SQL query only defines *what* to do, but it does not specify *how*. That means, although the domain-specific language has a strictly defined syntax, its semantics is subject to the actual implementation. As a consequence, most of today's SQL-based database management systems perform query optimisations beforehand, or

[11] Quote from "Refactoring: Improving the Design of Existing Code" [38, p. 15]. Martin Fowler is a British software developer, who spent most of his career in the United States. He popularised many modern software engineering principles and is one of the authors of *The Manifesto for Agile Software Development*.

[12] MySQL, https://www.mysql.com/, is one of the most widely-used open-source databases in the world. It is utilised by many popular websites, including Facebook, Twitter, and Wordpress.com.

dynamically decide when to use an index. Both aspects are not encoded in the query, as they do not affect its result, but only its performance.

However, SQL is just a single component of the relational database management system MySQL, providing a standardised interface both for users as well as for access by various general-purpose programming languages. Under the hood, MySQL is implemented in C and C++ [23]. For import and export, it supports several data exchange formats, e. g., comma-separated values (CSV).

It is a challenge in the process of software development to combine all these languages into a single product. Apparently, they differ not only in their syntax, but also in their underlying mental model of computation. As can be seen, SQL mostly follows a declarative approach to express the computation's logic without describing its control flow. On the other hand, it comes with procedural descriptions for control flow and exception handling, similar to C++'s imperative programming model that precisely express the commands for the computer to perform.

The different semantics of the considered languages is only one side of the coin. Another challenging question is how to integrate them into a single software component, and in particular how to support their specific syntaxes. It is often neither possible nor desirable to split a large codebase only based on the used programming language, with separate files for every used language, Instead, it is for instance common to put SQL statements right beside the code of the host language that processes the returned data rows. This can be done in various ways. Database statements in SQL are traditionally embedded as a string right next to the code of the host language. However, building large SQL statements from smaller, string-based blocks often comes with the risk of SQL injections. Since user input has to be properly sanitised anyway, there is a shift towards the use of fully-featured layers for object-relational mapping (ORM), thus integrating SQL in a way that more resembles the host language's syntax.

In this chapter, we consider the problems that arise when integrating different languages, as briefly sketched in this introduction. Firstly, in Section 4.1, we clarify the meaning of *domain-specific languages* and other terms in the context of this work, as there are various definitions in literature. The two major approaches to the integration of existing DSLs into a host language are described in Section 4.2. Though the work at hand is focussed on using Prolog as the host language to integrate various external DSLs, the opposite direction raises similar research questions. Since Prolog is often regarded as a domain-specific language for logic programming, there are many systems of other languages that either integrate or connect to (a subset of) Prolog. Section 4.3 provides an overview of these existing solutions, and how they handled the omnipresent impedance mismatch of the different programming

paradigms. Finally in Section 4.4, we summarise the existing approaches when using Prolog vice versa as the targeted host language. This section builds a bridge to the Chapters 5 and 6, which discuss in more detail the integration of DSLs in Prolog internally and externally.

4.1. Terminology

In literature, there are various definitions and notions for languages, programming languages, and domain-specific languages. In this section, we introduce the various meanings, and put them into context for the work at hand.

Languages. In this thesis, we consider the use of Prolog as a *host language* to integrate other computer languages. In literature, host languages are also called *hosting languages*, or *base languages*.

The integrated computer languages could be of various kinds:

- *Programming languages* (e. g., Assembler, PHP, Prolog) are formal languages to specify machine instructions. Their grade of abstraction might vary.
- *Markup languages* (e. g., HTML, MathML, LaTeX) extend a document by syntactical annotations used to format the contained text.
- *Query languages* (e. g., SQL, GraphQL, XPath) are used to specify queries on information systems.
- And others, like those for style sheets (e. g., cascading style sheets CSS), and for modelling (e. g., unified modeling language UML).

In the context of this thesis, we refer to all these kinds of computer languages simply as *languages*. In particular, we do not distinguish languages that are executable from those which are not. This follows Prolog's core principle that data and logic are not separated and share first-order logic predicates as their unified representation. As a consequence, the main focus of this thesis is on the *syntax* of the integrated language. Its semantics can be defined just as a normal Prolog program, possibly written as a Prolog meta-interpreter. For the discussion of connecting a language with Prolog, programming languages designed to contain executable source code are not treated differently from markup languages.

Domain-Specific and General-Purpose Languages. Once we build a bridge between Prolog and the integrated language, it can be directly used from within the Prolog program, extending Prolog's original syntax, expressiveness, and adding support for new data formats. We therefore also refer to the integrated language as a *domain-specific language* (DSL) in the context of this work. Note that this is different to some literature that regards a DSL as a computer language which is specialised to a particular application domain, in contrast to a *general-purpose language* (GPL). Since there is no exact and standardised definition of domain-specific languages, the question what exactly is a DSL is subject to debate. In [120], the following definition is proposed:

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focussed on, and usually restricted to, a particular problem domain.

This definition focusses on the problem domain, instead of the language's expressiveness or kind. However, the line between domain-specific languages and general-purpose languages is not always sharp, as in practice a language may have features that make it well suited particularly for a special application domain, but might simultaneously be also applicable more broadly. Conversely, a language may in principle be capable of various areas of application, but practically used only for a specific domain of problems. For instance, JavaScript was originally created by Netscape to allow dynamic behaviour of websites after the page was loaded in the web browser, but has since become a general-purpose language used both on client-side and server-side. In contrast, LaTeX is a Turing-complete language, and could consequently be used for any task, but is in practice only applied as a markup language almost exclusively in the research community.

As can be seen, for an existing language it is hard to classify it as domain-specific or general-purpose language. Often it can be done only in retrospective. Therefore, we summarise all languages, whose integration in a host language was already considered, simply as DSL. With this term, we do not imply restrictions to their application area or expressiveness, which is often the case when referring to DSLs in literature. Consequently, our presented considerations are equally applicable to (possibly a subset of) a GPL, though its syntax would be fairly restricted.

Internal and External DSLs. There are two approaches when integrating a DSL in a host language. Firstly, the DSL can be defined using only the syntactical features of the host language. In most imperative programming languages, the means are often limited to clever naming of functions, procedures, and variables. Others allow to

define macros to extend the programming language's syntax. Many declarative programming languages on the other hand provide a more flexible syntax. For instance, a DSL in Prolog can be defined as a subset of the host language by specifying appropriate operator definitions. We call a language that is composed by Prolog tokens forming valid Prolog terms an *internal DSL*. On the other hand – which describes the second approach –, the integrated language needs to be parsed and interpreted using standard compiler tools. Depending on the DSL's complexity, this requires the use of a lexer, parser, and/or interpreter. We refer to a language that is integrated using this classical compiler construction approach as an *external DSL*.

Note that this distinction is only based on the used integration technique. Though in this thesis several criteria to decide whether to implement a language either as internal or external DSL in Prolog are presented, this separation is not exclusive: any language that has been modelled internally could also be integrated by defining an appropriate fully-featured parser. As a consequence, following this definition, the set of internal DSLs is a strict subset of external DSLs. To put it the other way round, not all external DSLs are valid subsets of their host language and thus can be defined internally, where on the other hand every language can be integrated externally.

Grammar. Using an external DSL always requires a step to translate the integrated document into a data structure of the host language that represents its content. In the case of Prolog, the external DSL has to be translated into native Prolog terms. Though the embedded document can be processed by any means of the host language (e. g., ordinary Prolog predicates), this parsing step is typically based on a specification of the external DSL in the form of a formal grammar. We use the term *grammar* as an umbrella term for any kind of formal language specification, thus covering both context-free and context-sensitive grammars. It also does not imply a corresponding parser generator or concrete evaluation mechanism, though the grammars could always be implemented using Prolog's formalism of definite clause grammars, as introduced in Section 6.2.

Embedding. Terminology related to the integration of a DSL in a host language can be confusing, as the term "embedding" has been overloaded in three ways. Firstly, it is often used for the overall integration process – without a special focus on the phases of language design, implementation, and connection –, i. e., embedding just describes the idea of connecting two languages in a single software project. In addition, the term "embedding" is also used to describe the following two actual integration techniques. In Section 4.2.1, we define it as a means to reuse and compose

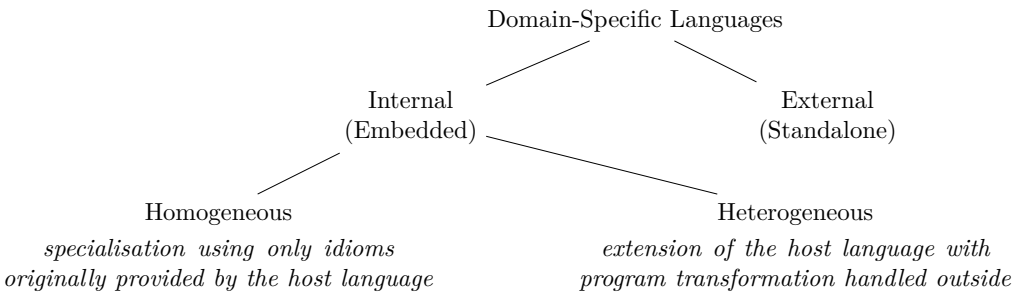

Figure 4.1: Taxonomy of domain-specific languages.

only idioms of the host language in a particular way that fits to the DSL's language design. Languages implemented this way are sometimes also explicitly called domain-specific embedded languages. In literature, however, the term "embedding" also covers the definition of a DSL as an extension or specialisation of the host language, which usually require adaptions on the host language's compiler. To differentiate both techniques, the terms *homogeneous embedding* and *heterogeneous embedding* have been used. In Figure 4.1, we provide an overview of the taxonomy of DSLs, following [117, 125].

Because the three meanings of "embedding" are a source of possible confusion, we use the term "embedding" only for the integration technique we present in Section 4.2.1, i. e., for internal domain-specific languages.

4.2. Integration Techniques

There are two major approaches to integrate a DSL into a host language: embedding and compilation. In the following, we give a more detailed overview of both techniques. It is the foundation for our discussion on the integration of DSLs with respect to Prolog in Chapters 5 and 6.

4.2.1. Embedding

For novel DSLs, the easiest approach is to start with the means provided by the host language, i. e., to create the integrated language as an extension of (a restricted version of) the host language. By only using the syntactic mechanisms of the host language – e. g., with user-defined functions and operators –, idioms of the embedded language are expressed, thus being valid terms in both the integrated and the host language. As a consequence, there is no need to define a separate parser, since the compilation is left to the host language's parser. This relieves the burden of the

programmer to keep the grammar, parser, and interpreter in sync. The result is an implementation infrastructure that supports rapid prototyping of newly defined domain-specific languages. The main advantage of this approach is that all features of the host language remain available and can be used as-is, i. e., they do not need to be re-implemented from scratch.

Though there are few tools dedicated to the design and implementation of a domain-specific language using this approach, all software development tools originally written for the host language can be applied, including syntax highlighting, auto-completion, and static code analysis. Code fragments of the external language can be put directly beside code of the host language. Its usage is not different than working with the host language, as they share their run-time environment. Only if the host language requires a compilation step, it is also needed for the integrated language. However for greater control on the execution of the internal DSL, applications sometimes ship with a self-defined interpreter. Since it is built using same language as the host and domain-specific language, it is called a *meta-interpreter*.

Languages following the implementation technique as an internal DSL are called *domain-specific embedded languages* [52] or *embedded domain-specific languages* [53, 65], or EDSL for short. A major drawback of embedding is its limited expressiveness and syntactical flexibility. Though the DSL has been adapted for the application domain, the embedded idioms have to be valid in the host language, and are therefore restricted to its syntax. Therefore, in many cases the optimal domain-specific notation has to be compromised to fit the limitations of the host language. Given an existing external data format, chances are low that it can be represented as a subset of the host language.

A very basic yet instructional example for the extension of a host language to enhance the program's readability in Algol 60 is given in [34]:

> And there was a little-known syntactic variant in the Algol 60 official syntax that encouraged a more readable form for made-up procedures. This allowed a comment in a procedure call to be replaced by the following construct:
>
> ```
> ): <some comment> (
> ```
> Algol 60
>
> and this would allow `[ for (i, 1, 10, print(a[i])) ]` to be written as follows [...]:
>
> ```
> [ for (i): from (1): to (10): do ( print(a[i])) ]
> ```
> Algol 60
>
> which looks a lot like the Algol base language but done as a meta-extension by the programmer for the benefit of other programmers.

Other examples for the approach to create a DSL by extending the host language's syntax are *Frob* and *Fran* [98], two domain-specific extensions of Haskell to control a robot and describe reactive animations, and the PIC-like drawing language *FPIC* [60] which is embedded in ML.

4.2.2. Compilation

Compilation is the classical approach to add support for a new language in a particular host language. Based on a given syntax definition – often in the form of a context-free grammar –, a dedicated method for parsing is written. One can use standard compiler tools, like the Java-based parser generator ANTLR [93]. The resulting parser creates a corresponding representation using the host language's data types, which can be later used for interpreting the encoded information and instructions. Therefore, compilation is usually done in combination with interpretation. The used programming languages can differ in all of these steps.

Following this approach, the external language can be put in a separate file and thus be read in from the host language. To what extent code fragments can be embedded directly into the host language's source code depends on the host language's capabilities to handle multi-line strings, and requirements with respect to escaping symbols of the embedded language that are otherwise special characters in the host language.

The main advantage of implementing a language this way is a great flexibility. There are no restrictions to the embedded language's syntax, so no concessions regarding the language's notation and primitives have to be made. Vice versa, this technique can be used to add support for any existing language and data format. Having a dedicated parser and interpreter also simplifies the error handling, as missing primitives can be acknowledged by self-defined error messages. In addition, optimisations and static analysis can be made at compiler level. The resulting program therefore come with a reasonable performance.

Clearly, this great flexibility comes at a price. Even with assisting tools, building a compiler and interpreter from scratch requires in-depth knowledge about compilers, state machines, language theory, and grammars. Aside from this, it is hard to reuse existing grammars and parsers, as there is no standard in how to extend, restrict, and combine those describing different DSLs. Once written, modifications to a grammar, the corresponding parser, or the interpreter require changes in all of the other parts, resulting in a tightly coupled software system with a lot of challenges regarding maintenance and continuous development. Designing and implementing languages this way is difficult and resists evolution [52].

There are no canonical examples of DSLs that are integrated into a host language using this compilation approach, since all general-purpose languages provide interfaces to work with external data formats. On the other hand, most data exchange formats were not designed with an executing programming language in mind, so they were originally not intended to be a strict subset of some host language.

4.2.3. Preprocessing and Extensible Compilers

The decision to whether implement a DSL internally by embedding or externally using compilation depends on both the syntax of the integrated DSL as well as of the host language. It can be observed that there are more EDSLs defined internally in *small-syntax languages* like Lisp, Haskell, and Smalltalk, as their syntax is less restricted and the number of predefined keywords is limited. *Large-syntax languages* like Java and C++ on the other hand typically require techniques such as preprocessing or extending the compiler [29, 73]. These are techniques that can be used together with embedding or compilation.

Preprocessing or Macro Processing. In a preprocessing step, the DSL's language constructs are translated into statements of the host language. In contrast to the compilation approach, the DSL is defined as an extension to the host language, so only some parts have to be replaced in order to be executable. The main advantages are simplicity and expressiveness, as it does not require a complete language definition or compiler implementation, while still all of the host language's idioms can be used. On the other hand, static checking and optimisation cannot be done at the domain level. Since the preprocessing step is often implemented by string replacements using regular expressions, it is error-prone, and so is the generated code.

Extensible Compiler or Interpreter. Instead of relying on string-based replacements, the preprocessing step can be integrated in the host language's compiler. This way, the DSL is still defined as an extension to the host language, but type checking and optimisation is possible and performed as part of the host language's build step. Some compilers support this as a means for code composition [115], e. g., the ANSI C compiler framework *Catacomb*. The Tcl [92] interpreter is another good example, as it has been extended for dozens of domains.

Tradeoffs. Both the preprocessing and extensible compilers are common techniques when extending large-syntax languages. However, they lose some of the advantages that come with a pure language embedding approach, e. g., the compatibility and

adaptability of existing tools originally designed for the host language, and the lack of additional build steps. After all, embedding DSLs in these languages might result in more problems than are solved. “Extending Java has at times been a frustrating experience”, the authors of the Java testing framework *jMock* [39] conclude in their EDSL experience report [40].

4.3. Prolog as a DSL in other Host Languages

Sometimes, Prolog is also regarded as a DSL, tailored to the specific application domain of logic programming, knowledge representation, and deduction. In this thesis, we do not elaborate on the discussion of whether Prolog is a DSL or GPL, and instead just use Prolog as our preferred host language to embed other languages. However, as the embedding of Prolog in some other GPL raises similar research questions as discussed in our work, we give a short overview in this section of existing systems that integrate Prolog in various programming languages.

All presented approaches have a common goal: combining the logic-based programming paradigm with another language to create a system that is more expressive than its parts. However, the major challenge is the different execution model (imperative vs. declarative, often with required support for backtracking). As a consequence, systems have to make architectural decisions both for a natural integration regarding the syntax of Prolog and the host language, as well as the more fundamental discussion of integrating different semantics. This often comes with interoperability problems due to the paradigmatic gap.

Existing connectors to Prolog from other languages usually stick with the host language’s syntax to integrate Prolog idioms. The syntax of Prolog is therefore modified in order to fit into the syntactical requirements of the host language, or the Prolog parts have to be put in separate files. In both ways, Prolog is effectively treated as an external DSL. Besides syntax incompatibilities, this approach is often justified by better developer experiences, as programmers are usually only familiar with the host language’s syntax.

4.3.1. Java

Combining Prolog and Java has been subject to research for decades. [90, Sec. 2] gives an overview of systems that combine these two languages.

4.3.1.1. SWI-Prolog's Prolog/Java Interface JPL

Because it ships with SWI-Prolog, JPL [111] is one of the most well-known and sophisticated interfaces between the two languages. It provides a two-way binding, based on the Java Native Interface (JNI) to connect to a Prolog engine, and the Prolog Foreign Language Interface (FLI) [136, Sec. 12] which allows to call foreign language functions as Prolog predicates. To integrate Prolog into Java, JPL provides special classes that mimic the syntax of Prolog. For instance, the predicate call `?- consult('database.pl').` can be stated using JPL as follows:

```java
Query q1 = new Query(
    "consult",
    new Term[] {new Atom("database.pl")}
);
```

Prolog snippets cannot be put verbatim next to Java code. Instead, Prolog can be used only as an external DSL. Though JPL provides bidirectional classes and methods to directly interact with Prolog terms, queries, and answers, their usage leads to verbose Java code for simple compound Prolog terms. In addition, this kind of integration requires the developers to have knowledge and deep understanding of both language's internals.

4.3.1.2. CAPJa Connector Architecture

In the integration framework *Connector Architecture for Prolog and Java* (CAPJa) [89, 90], the mapping from Prolog terms to Java classes and vice versa is done fully automatically, and can be further manipulated using Java class annotations. This way it is possible to define classes, terms, and their mappings as a data interchange format once, and then work with plain old Java objects. For instance, in Listing 4.1, two persons are described using a self-defined `Person` class. With CAPJa's default mapping, they are mapped to the compound Prolog term `person/3` (l. 14). `@PlView` class annotations would allow to modify this mapping of `Person`, e. g., to use the person's identifier in the list of children instead of nesting the terms.

The CAPJa framework is completely implemented in Java and imposes no modifications to the Java Virtual Machine or Prolog. Under the hood, CAPJa ships with an extensible system of gateways that provide connectivity with various Prolog systems, so it can be easily changed without further code modifications. The gateway uses a text-based data exchange format between Java and Prolog, called *portable Prolog interface* (PPI) [91] for Java. It is based on standard streams `stdin`, `stdout` and `stderr`, and serialised Prolog terms to communicate between Prolog and Java.

Listing 4.1: Default object mapping in CAPJa. Condensed code example from [90, Sec. 3].

```java
class Person {
    private int id;
    private String givenName;
    private String familyName;
    private Person[] children;
    // ... constructor, getter, setter
}

Person p1 = new person(1, 'Homer', 'Simpson');
Person p2 = new person(2, 'Bart', 'Simpson');
p1.setChildren(new Person[]{p2});

// corresponding Prolog term using default mapping:
//   person(1,'Homer','Simpson',[person(2,'Bart','Simpson',[])])
```

An advantage of this integration approach is that Prolog and Java developers can work separately on their parts, remaining with their preferred language: Java for object-oriented development, and Prolog for rule-based components. On the other hand, CAPJa solely focusses on the interaction and interfaces of the two languages, rather than provide a seamless integration of their different syntax. This way, Prolog is treated again as an external DSL.

4.3.1.3. Constraint-logic Object-oriented Language Muli

The constraint-logic object-oriented programming language *Muli* [28] instead is realised in the form of an extensible compiler. It extends Java by logic variables and encapsulated, constraint-based search. Though it is not focussed on the integration of Prolog, it adapts several concepts known from logic- and constraint-based programming languages, resulting in a system that resembles SWI-Prolog with its *library(clpfd)* for constraint-logic programming over finite domains, just extended for object-oriented programming.

Muli is implemented as a minimal language extension to Java 8. It is a superset of Java with additional constraint solving and nondeterminism features. Consequently, every Java program is also a Muli program that can be compiled and executed by Muli. Regarding Muli's syntax, only a single keyword is added to Java's original grammar. Variables can be marked as logic variable using the `free` keyword, e. g., in `int x free`. This syntactical extension of Java's compiler is fairly simple and requires only minor modifications in Java's EBNF rules for declaring a field; it is

Listing 4.2: XML Schema to specify an object mapping as used by GOOMN.

XML Schema

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person" type="Person" />
  <xs:complexType name="Person">
    <xs:sequence>
      <xs:element name="givenName" type="xs:string" />
      <xs:element name="familyName" type="xs:string" />
      <xs:element name="children">
        <xs:complexType>
          <xs:sequence name="person"
            type="Person" maxOccurs="unbounded" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

described more in detail in [29]. The more challenging part here is to define and implement the operational semantics of Muli programs that use logic variables, as the free variables and the constraint solving problems introduce nondeterminism.

4.3.2. Python

The *Python/Prolog Database Connectivity* (PyPlC) [9] toolchain aims to provide a portable interface for the interpreted programming language Python. With PyPlC, Prolog structures are converted into corresponding object-oriented structures that can be easily processed by Python and vice versa. Similar to CAPJa, the mapping from Prolog terms to classes of the host language can be generated automatically.

In regard to portability, PyPlC resembles the ideas of the famous *Open Database Connectivity* (ODBC) interface for accessing relational database management systems. This standard application programming interface (API) is implemented by all major SQL systems. In addition, the API is supported by many programming languages and libraries to connect to a database system using a standardised set of instructions. Therefore, in contrast to CAPJa's Java-based mapping of Prolog terms to classes, Bodenlos et al. define a language-independent, XML-based format to describe Prolog data structures in [9]. This format, which is called *general object-orientated mapping notation* (GOOMN), contains all information about data types and structures of compound terms. For each component of a Prolog structure, the GOOMN specification provides a meaningful name, as well as a generic description

of the type. Once a Prolog database is described this way, it can in principle be mapped to any object-oriented programming language due to its generic nature. Listing 4.2 shows an example XML Schema that describes the `Person` class we used for CAPJa (cf. Listing 4.1).

PyPlC creates a Python mapper class for each XSD type specified in the XML Schema and integrates it into the PyPlC module. By changing the GOOMN notation, the mapping process is completely customisable: for instance, the software developer can decide whether a component should be mapped or not. Since it is possible to pre-compile the mapper classes, the processing speed can be significantly increased.

To still benefit from the advantages of the integrated logic programming system, the query processing and data storage in PyPlC is left to the connected Prolog interpreter, usually SWI-Prolog. Therefore, and to follow the example of ODBC, PyPlC offers a Python/Prolog query language, called PyPlQL. It is inspired by the query language JPQL, which is shipped with CAPJa. PyPlQL is an internal domain-specific language in Python, influenced by corresponding data structures of the two languages of interest, Python and Prolog. Simply put, a PyPlQL query is a Python function, whose parameters declare the required objects and whose function body describes the conditions to be fulfilled by a Boolean expression. Such functions are not executed as they are; instead, they get transmitted to PyPlC, which transforms and passes them on to the Prolog instance. The successful variable binding gets transformed and returned again as a Python object. Listing 4.3 gives an example on how to use PyPlC and PyPlQL to load an externally defined Prolog fact base `database.pl` with the GOOMN mapping `database_goomn.xml` and ask for all persons with a family name of “Simpson”.

4.3.3. JavaScript

Measured against distribution and popularity, JavaScript is currently one of the most popular programming languages. Douglas Crockford, who developed the JavaScript Object Notation (JSON), once stated that every personal computer in the world had at least one JavaScript interpreter installed on it and in active use [27]. In addition, JavaScript is already a popular target language for compilation, as it accounts for more than 300 languages that compile to it.[13] By integrating logic-based programming languages to JavaScript, one benefits from this broad distribution of run-time environments.

[13] List of languages that compile to JavaScript, https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js, document revision February 21, 2021.

Listing 4.3: Load and query a Prolog database in PyPlC. Adapted example from [9, Sec. 3].

```python
pyplc = Pyplc('swipl') # create an instance of PyPlC

# import the Prolog database and the corresponding GOOMN
pyplc.import_database('database.pl', 'database_goomn.xml')

# define and execute query
def person_query
  (p : Person):
  p.familyName == 'Simpson'
res = pyplc.retrieveAll(person_query)

# print all retrieved persons
for p in res:
  print('Given Name: ' + p.givenName)
```

JavaScript and Prolog have a property common: both traditionally lack support for multi-line strings, but have been recently enhanced in this regard by new language idioms. Since version 6.3.17, SWI-Prolog ships with quasi-quotations, which we introduce in detail in Section 6.1. JavaScript on the other hand supports *tagged template strings*. They have been introduced to JavaScript in the standard ECMAScript 2015 (formerly ECMAScript 6) [141]. They are of the following basic form:

```javascript
tag`Content`
```

For instance, a multi-line HTML snippet can be embedded as an external DSL in JavaScript as follows:

```javascript
html`<h1>Hello, ${name}!</h1>
     <p>How are you today?</p>`
```

Similar to quasi-quotations in SWI-Prolog, a tagged template string of the previous form invokes a function call of the user-defined function `tag` (called the *template handler*), i. e., in this example `html(content)`. Expressions from the outside-world can be referred to in the embedded document using the `${...}` notation. To process the given string content, it is split by the embedded expressions:

```javascript
let name = 'Alice'
function html(strings, ...values) {
  return strings[0] + values[0] + strings[1]
}
html`<h1>Hello, ${name}!</h1>` // returns '<h1>Hello, Alice!<h1>'
```

Listing 4.4: Rules in CHR.js to calculate the greatest common divisor. Adapted example from [78, Sec. 5].

```
var CHR = require('chr') // load CHR.js                                    JavaScript
var chr = new CHR() // initialise CHR runtime, e.g., constraint store
chr`
  gcd(0) <=> ${ console.log('Finished') }
  gcd(N) \\ gcd(M) <=> 0 < N, N <= M | gcd(M-N)`
```

Unlike Prolog, JavaScript has no built-in grammar formalism to modify the parser that is used for the embedded document. Instead, third-party libraries have to be used to parse the given content strings and generate the corresponding replacement. Similarly to other languages providing features like quasi-quotations and tagged template strings, there is no standard like the DCGs known from Prolog for JavaScript.

4.3.3.1. CHR.js JIT Interpreter and AOT Compiler

In [78], we use tagged template strings to embed the constraint-logic programming language Constraint Handling Rules (CHR) [41] as a domain-specific language in JavaScript. Listing 4.4 presents the classical CHR example of computing the greatest common divisor of an arbitrary number of integers specified in `gcd/1` constraints. Regarding the traditional syntax of CHR, the embedding in JavaScript requires only minor modifications. Firstly, since in JavaScript the backslash `\` is an escape character in strings, it has to be escaped by a second backslash. Secondly, instead of explicitly declaring CHR constraints in order to differentiate them from built-in constraints (i. e., operators and functions provided by the host language JavaScript, for instance `console.log()`), we rely on the expression embedding using the `${...}` notation.

CHR.js provides both a just-in-time (JIT) interpreter and an ahead-of-time (AOT) compiler. The JIT interpreter allows to embed CHR code snippets in JavaScript using tagged template strings as presented in Listing 4.4. This allows to put the CHR rules directly next to the code of the host language JavaScript, but relies on a separate parser, i. e., CHR is treated as an external DSL. The AOT compiler on the other hand extends the JavaScript grammar – the CHR rules are treated as first-class citizens, so it is not needed to enclose them by tagged template strings. In short, CHR is embedded internally following the preprocessing approach we introduced in Section 4.2.3.

Listing 4.5: Initialisation of Tau Prolog with a Prolog fact base.

JavaScript

```
1  var pl = require('tau-prolog') // load Tau Prolog
2  var session = pl.create() // create a Prolog session
3
4  // ?- consult('database.pl').
5  var q1 = new pl.type.Term(
6    'consult',
7    [new pl.type.Term('database.pl', [])]
8  )
9  session.query(q1)
10
11 // alternatively load data in a tagged template string
12 session.consult`
13   person(1, 'Homer', 'Simpson').
14   person(2, 'Bart', 'Simpson').`
```

In order to parse the JavaScript source code, we formalised the syntax of the CHR rules in [76, Sec. 4.3] using a Parsing Expression Grammar (PEG) [37] and its JavaScript parser generator PEG.js.[14] The JavaScript constraint solver created by the AOT compiler is of competitive performance [78].

4.3.3.2. Tau Prolog Interpreter

Tau Prolog [99] is a client-side Prolog interpreter fully implemented in JavaScript. Similar to Java's JPL (cf. Section 4.3.1), it provides JavaScript prototypes to construct Prolog terms. These are similar to classes known from object-oriented programming languages. Consequently, the JavaScript code to create a simple goal in Listing 4.5 (ll. 4–9) resembles those of Jpl. In addition, Tau Prolog also makes use of quasi-quotations to embed Prolog code verbatim in JavaScript (ll. 11–13). These strings are transformed internally into an abstract syntax trees, which forms the basis for the in-browser execution. In this regard, Tau Prolog is implemented similar to the CHR.js JIT interpreter, with Prolog treated as an external domain-specific language.

4.3.4. Haskell

The previously presented systems and frameworks integrate Prolog in an object-oriented programming language. These usually come with a large number of pre-

[14] PEG.js, https://pegjs.org/, is a parser generator for JavaScript. Its syntax is similar context-free grammars and Prolog's definite clause grammars.

Listing 4.6: Systematic translation of the predicate `append/3` to Haskell, following the definition of the logic program as formula F' in Section 2.5.

```haskell
append(x, y, z) =
  (x ≐ nil & y ≐ z) ||
  (exists (\e -> exists (\l -> exists (\m ->
    x ≐ cons(e, l) & z ≐ cons(e, m) & append(l, y, z) ))))
```

defined keywords and great constraints regarding the DSL's syntax. Consequently, none of these systems rely on the embedding of Prolog; only Muli uses an extensible compiler for modifications of the Java programming language. Haskell one the other hand is a small-syntax language, and therefore comes with a higher flexibility regarding the embedding of DSLs (cf. Section 4.2.3). Its syntax is nevertheless not compatible with Prolog. Existing approaches to embed Prolog in Haskell are mainly focussed on how to close the semantic gap of both languages and programming paradigms.

To overcome syntactic differences, the authors of [113] define an algebraic data type `Predicate` for predicates, `Term` for terms, and various other types, e. g., to hold the list of substitutions in a unification. In addition, four Haskell functions need to be defined: `(&)` and `(||)` denote conjunction and disjunction of predicates, `(≐)` forms a predicate expressing the equality of two terms, and `exists` expresses existential quantification. These four functions suffice to translate any pure Prolog program.

Based on these functions, the classical `append/3` predicate can be expressed in Haskell as presented in Listing 4.6. It strictly follows the definition of the formula F' in first-order logic we specified in Section 2.5. Lines 1–2 correspond to the rule F'_1, which is an alternative representation of the fact F_1. Lines 3–6 implement the recursive rule F'_2 with explicit existential quantification of the variables E, L, and M.

The program transformation from source code originally written for Prolog to the data types and functions in Haskell can be done automatically:

1. Since variables in Haskell are denoted by lowercase letters, they are renamed accordingly.

2. Prolog's unification is replaced by the newly defined `(≐)` function.

3. Local variables in a rule's body have to be explicitly existentially quantified using the `exists` function and lambda expressions (l. 3).

4. Conjunctions and disjunctions are replaced by the functions `(&)` and `(||)`. Prolog rules with the same predicate in the head are combined into one disjunction, with unifications originally stated in the rule's head being transformed into logical preconditions.

Similar to Prolog, Haskell allows to define new operators with their precedences. Nevertheless, Haskell maintains only the ten precedence levels from 0 to 9, in contrast to Prolog's 0 to 1200. For instance, in Listing 4.6 we use the infix operators `&`, `||` and `≐`. In contrast to Prolog, Haskell does not support postfix operators, reducing its capabilities to embed DSLs internally.

4.3.5. Julia

Julia [8] is a high-level, high-performance, dynamic programming language, which is mainly used for numerical analysis and computational science. It is a faster alternative to MATLAB, with a compiler written in C, C++, and Scheme.

The Julia package *Julog.jl*[15] provides a `@julog` macro that can be used to define logical terms and Horn clauses in a Prolog-style syntax. Listing 4.7 shows how to implement and query the predicate `append/3`. It looks fairly similar to the original Prolog code, as in particular the notations for predicate symbols and variable symbols are the same as in Prolog, i. e., terms look the same in Prolog and Julog. With respect to the definition of clauses, only minor changes are needed for Julog code to be compatible with the original syntax of Prolog:

- A clause does not end with the full stop.
- Facts have to be stated with an explicit rule body of `true`.
- `<<=` is used instead of Prolog's infix operator `:-/2`$_{\texttt{ISO}}$ to separate the rule's head and body.
- Conjunctions are denoted by the `&` instead of Prolog's infix operator `,/2`$_{\texttt{ISO}}$.
- Negation is denoted by the prefix `!` instead of Prolog's operator `\+/1`$_{\texttt{[C.2]}}$. The cut is instead available as an explicit predicate `cut`.

Julog supports forward-chaining proof search (i. e., bottom-up evaluation) using the `derivations` function, as well as the classical backward-chaining proof search (top-down) via SLD resolution with the `resolve` function. However, the SLD resolution is implemented as a breadth-first search instead of Prolog's depth-first approach,

[15] Julog.jl, https://github.com/ztangent/Julog.jl, Apache License 2.0. The code example of Listing 4.7 is based on version 0.1.11, September 2021.

Listing 4.7: Definition and usage of the predicate `append/3` in Julia.

```julia
clauses = @julog [
  append([], Y, Y) <<= true,
  append([E | X], Y, [E | Z]) <<= append(X, Y, Z)
]
query = @julog [append(X, Y, [a,b])]
satisfiable, substitution = resolve(query, clauses)
# satisfiable = true
# substitution = 3-element Array{Any,1}
#   {X => [], Y = [a,b]}
#   {X => [a], Y = [b] }
#   {X => [a,b],Y = [] }
```

resulting in a different semantics for partially solved solutions and in particular for the application of the cut. There is no equivalent for Prolog's operator `;/2`, so disjunctions have to be made implicit by stating alternative rules.

4.3.6. The Prolog Transport Protocol

Integrating Prolog into another language, particularly when following another programming paradigm, often comes with several conceptual and technical difficulties regarding different data structures and execution models. This question is similar to the well-known *object-relational impedance mismatch*. It can be encountered when a relational database management system is being served by an application program that was written in an object-oriented programming language or style. There, objects or class definitions must be mapped to database tables defined by a relational schema. Similarly, when integrating Prolog, the backtracking-based nondeterminism has to be mapped to pagination, and logic variables to built-in data types.

This mapping is similar for all imperative programming languages, since conversations with a Prolog instance follow a communication protocol that has been formalised as the *Prolog Transport Protocol* (PLTP) in [70, Sec. 4]. It is based on the Prolog 4-port model [14] (cf. Section 7.3.1), extended with exceptions and I/O. It follows a request-reply format that makes it suitable for imperative programming languages and transport layers. Therefore, it lays the foundation for remote procedure calls in Prolog. SWI-Prolog uses PLTP for the inter-thread communication of distributed Prolog applications, and to provide the web-based *Pengines* interface. The latter allows a Prolog server and JavaScript client to communicate via PLTP over HTTP ($\text{PLTP}_{\text{HTTP}}$).

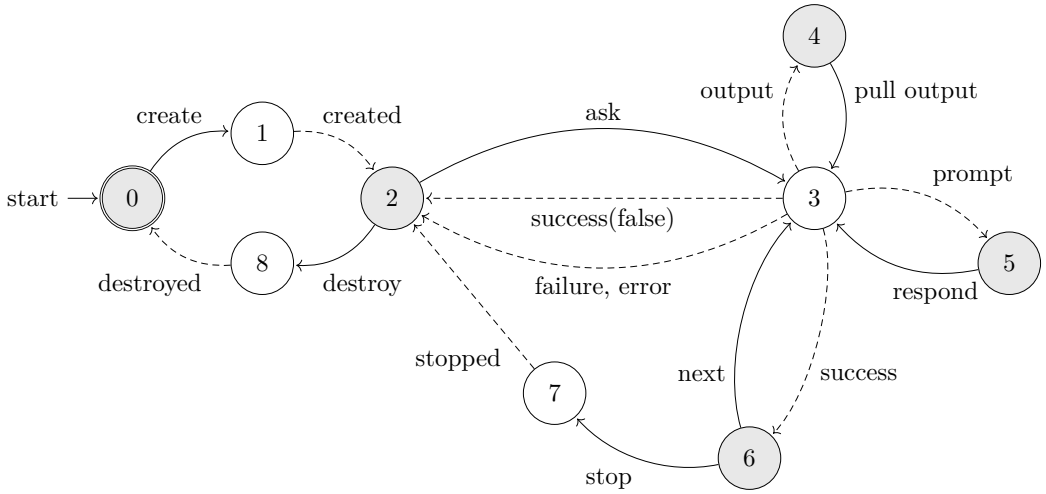

Figure 4.2: Finite-state machine for the Prolog Transport Protocol, based on [70].

PLTP is built on top of a finite-state machine. Figure 4.2 shows an adapted version from [70]. Since PLTP is based on a request-reply communication protocol, normal edges describe requests, dashed edges replies; the states are depicted accordingly, with states that require an action from the client filled grey, and white otherwise. The states are as follows:

- (0) Disconnected
- (1) Connecting
- (2) Idle
- (3) Running query
- (4) Query produced output
- (5) User interaction
- (6) Idle after query processing
- (7) Query stopped
- (8) Disconnecting

State (3) describes the four different outcomes when querying Prolog:

- (2) The query processing in stopped in case of an occurred error or failure, or if there is no (further) successful refutation in the SLD tree. In the toplevel, the user would be asked for a new query, denoted by `?-`.
- (4) An output via I/O is produced. In the toplevel, this output would simply be printed.
- (5) The user gets prompted for additional information via I/O, e. g., using `read/1`. This would be denoted by `|:` in the toplevel.
- (6) The computed answer substitution of a successful SLD refutation is given back, allowing the user to ask for another solution or to stop the query processing.

In the toplevel, this could be answered either by `;` (or alternatively `n` for "next") or `.` (or alternatively `a` for "abort").

With PLTP, the client component and Prolog server can be developed independently, and for various transport layers. Currently, SWI-Prolog is the only Prolog system with support for the protocol. Given that other systems like GNU Prolog or Tau Prolog also implement $\text{PLTP}_{\text{HTTP}}$ in the future, it would be possible to interchange Prolog backends. In this regard, PLTP is similar to the *Portable Prolog Gateway* (PPG) which is shipped with CAPJa [89, Sec. 7.3]. The PPG is based on the streams for standard input `stdin`, standard output `stdout`, and standard error `stderr`. Besides SWI-Prolog, it has been successfully tested with several Prolog systems such as the Ciao system [12, 50], GNU Prolog [32], the XSB system [116], and YAP [24]. The PPG therefore emphasises the feasibility of a system-independent communication protocol, and lays the foundation for a PLTP implementation based on standard streams, i. e., $\text{PLTP}_{\text{stdio}}$.

4.4. The Status Quo on the Integration of DSLs in Prolog

In the previous section, we provided an overview of existing systems that integrate Prolog in various other programming languages. The problem put the other way round is subject of research in this thesis. Before discussing Prolog's capabilities of integrating internal and external DSLs in Chapters 5 and 6 in more detail, this section provides an overview of existing DSLs that can be connected with Prolog. Additional examples of real-world programming languages with considerations regarding the integration of their code snippets into Prolog can be found in [130].

Internal DSLs. Not only because of its long history and ongoing development, SWI-Prolog offers a large software ecosystem that adds support for many popular computer languages and interfaces. But as of now, only a minor part is integrated in the form of an internal DSL. This is mostly because DSLs which have been designed without having Prolog in mind usually require syntax adaptions to be compatible with existing Prolog systems. As part of our work in Chapter 9, we discuss to instead slightly adapt the requirements to be valid Prolog syntax to allow more external DSLs to be defined and used internally.

In Section 4.3.5, we presented the integration of Prolog into the dynamic programming language Julia, which needs only modifications in the syntax of clauses, while terms can be represented similarly in both languages. This makes Julia a good

candidate for integrating the DSL internally in Prolog. In fact, SWI-Prolog's *library(pljulia)*[16] allows Prolog code to use an embedded instance of Julia and defines several self-defined operators to mimic Julia language features. The Prolog code though requires several modifications to be fully compatible with native Julia code.

The most common application of this technique to define appropriate Prolog operators is to create a completely new DSL, without the intend to use this internal DSL later in another host language. Popular examples are extensions of the logic programming language in Prolog, for instance *Constraint Handling Rules* (CHR) [41], and the probabilistic extension for Prolog *ProbLog* [62]. This also covers most testing libraries for Prolog like our *library(tap)*, which allow to simply specify the test cases and expected results in the form of a Prolog program. The declaration of tests can then be directly executed once the operators are defined as Prolog predicates as well. In Chapter 5, we discuss this technique in more detail.

Foreign Language Interface. Compared to the internal definition of a DSL, there are many more languages that are not integrated as a subset of Prolog. Instead, support for these languages is added either by defining a custom-built compiler, or by connecting Prolog to existing compilers written in C or C++. The latter can be done by calling foreign code from within Prolog.

In SWI-Prolog, it is possible to define a *foreign predicate*, which internally is a C or C++ function with the same number of arguments. The passed C/C++ and Prolog terms can be converted and passed along both systems by a bidirectional interface, effectively making all C and C++ programs available for Prolog. This allows to reuse existing C/C++-based compilers for external DSLs, and at the same time comes with a great flexibility and performance.

To use SWI-Prolog's *foreign language interface* [136, Ch. 12], foreign code must be put into a SWI-Prolog extension that is later included by `use_module(library(...))`. SWI-Prolog provides foreign interfaces to interact with existing C and C++ programs; Java could be connected via the bidirectional Prolog/Java interface JPL (cf. Section 4.3.1.1). The foreign extensions can be given by source as well as by providing only their executable binary. Packages containing foreign code are supported by most modern operating systems, e. g., with DLLs under Windows, or shared objects under Linux.

For instance, this approach is used for the implementation of *library(sgml)*, a parser for the Standard Generalized Markup Language (SGML). It is defined as a C-library

[16] *library(pljulia)*, https://github.com/samer–/pljulia, GNU GPL v3.

for SWI-Prolog that has been built from scratch, resulting in a lightweight and fast parser [138]. Since XML shares the same tree-model as SGML with some additional restrictions, *library(sgml)* forms the base for all XML processing tools in SWI-Prolog, as well as all its derived data formats. This also includes the Resource Description Format (RDF), a W3C standard for expressing meta data about web-resources, that is heavily used in semantic web applications in Prolog [139].

Another popular example of using existing foreign libraries to connect to external DSLs is *library(odbc)*,[17] SWI-Prolog's interface to ODBC, the Microsoft standard for Open Database Connectivity. This standardised interface is supported by a wide variety of SQL-based relational database systems. Under Linux, SWI-Prolog links to *unixODBC*;[18] under Windows, `odbc32.lib` is used. The provided Prolog predicates closely follow the ODBC API and provide therefore a low-level, yet fast access to the database management system.

External DSLs. Only a minor part of SWI-Prolog's packages that add support for other languages and data formats make use of Prolog's built-in capabilities to define parsers and compilers, in particular DCGs. One example is *library(protobufs)*,[19] which adds support for Google's Protocol Buffers. It is a language-neutral, platform-neutral, extensible mechanism for serialising structured data – similar to XML, but smaller, faster, and simpler. In SWI-Prolog, it is again based on a small C-library, but also uses DCGs to interpret the incoming data stream. The aforementioned *library(pljulia)* does similarly by converting the results from the embedded Julia instance back to Prolog terms with the help of DCGs.

Most current applications of DCGs in Prolog are related to natural language processing. In Chapters 6 to 8, we introduce our contributions that assist with the integration of external DSLs using these grammars.

[17] SWI-Prolog ODBC Interface, https://www.swi-prolog.org/pldoc/package/odbc.html.
[18] unixODBC, http://www.unixodbc.org/, is an open-source implementation of the ODBC interface, shipped with most versions of Linux and Mac OS.
[19] Google's Protocol Buffers Library, https://www.swi-prolog.org/pldoc/package/protobufs.html.

5

Prolog as a Host for Internal Domain-Specific Languages

Internal domain-specific languages are well designed by default.

— Marjan Mernik[20]

In the software development process, the solution to a problem requires to think about the data structures and how to process it. In this regard, we could change Robert Kowalski's famous equation from Section 2.1 to *algorithm* = *data* + *control*, with the same underlying idea: at best, the two parts, *data* (instead of *logic*) and *control*, are only loosely coupled, and can be changed independently. This allows to separate the two phases in the process of software development of modelling and processing data.

For large software this is already often the case, as classes and relationships are modelled before the actual implementation happens. However, the means to represent the data are always restricted to the concrete syntax and mental model of the used host language. Here, the homoiconic programming language Prolog has several advantages, since every Prolog data representation is also a valid Prolog program. The data can be simply stated as an executable program and further processed. This allows to first model the data with a flexible and hardly restricted syntax, and later define predicates that work on it, all in the same programming environment.

This approach is especially useful for rapid prototyping. Jan Wielemaker, main developer of SWI-Prolog, underlined this in a general advice regarding software development with Prolog:[21]

[20] Quote from the Preface to "Formal and Practical Aspects of Domain-Specific Languages: Recent Developments" [72]. Marjan Mernik is a Slovenian computer scientist and professor at the University of Maribor, whose main research interests include the intersection of programming languages and software engineering. From 2014–2014, he was Editor-in-Chief of the international journal *Computer Languages, Systems and Structures* (COMLAN).

[21] Answer by Jan Wielemaker in the SWI-Prolog Discourse forum thread "Represent Accounting Equation Using SWI-Prolog", https://swi-prolog.discourse.group/t/represent-accounting-equation-using-swi-prolog/1465/16, November 12, 2019.

> My favorite message: do *not* represent your knowledge/rules as executable Prolog. Instead, create a representation as Prolog terms that expresses the rules in a format your domain experts will like. Next, write a Prolog program that make them work. This can either be by interpreting them, compiling them to executable Prolog or even compile them into something else (JavaScript, C, ...).
> The first step is typically easy. The others are a bit harder, but can be delayed until you hit scalability issues.
> Note that such a format make[s] your domain experts happy and make[s] it much simpler to reason about your rules and find inconsistencies, produce explanations, etc.

Following this advice, it seems reasonable to use Prolog only as an intermediate language to express knowledge in a suitable way, often as a domain-specific language With Prolog's flexible syntax, the resulting data representation can be easily written and processed by domain experts and is a valid Prolog program at the same time. Thus, Prolog can be easily used either for direct data processing, or to convert the data in an external representation.

In Chapter 3, we presented the basics of programming in Prolog, with a syntax that is based on terms as the language's first-class citizens. In this chapter, we first introduce in Section 5.1 operators as functors as a means to write Prolog terms and thus programs in a more natural way. It relates to the data representation in the form of Prolog terms, as suggested in the quotation by Wielemaker. In Section 5.2, we introduce means to run data transformations at compile-time. Section 5.3 reconsiders the idea of interpreting data of this internal DSL representation with a Prolog meta-interpreter. As an example, we present in Section 5.4 the definition of an internal DSL for natural-language-flavoured if-then rules. They have been successfully applied to represent expert knowledge in the field of change management in organisational psychology. In Section 5.5, we discuss common features and differences between domain-specific languages and controlled natural languages. The chapter concludes in Section 5.6 with the definition of EBNF as an internal Prolog DSL. This formalism to describe context-free grammars is often used by syntax definitions of computer languages and thus provides the basis for parsing and serialising external Prolog DSLs as discussed in Chapter 6.

5.1. Operator Notation for Terms without Parentheses

In general, DSLs are used to formulate propositions about real-world entities. These can be identified by numbers, strings, and variables, which happen to be the smallest

units of all Prolog programs. Complex propositions can be composed using compound terms. As introduced in Section 3.1.1, compound terms normally begin with the functor name, which is an atom, followed by a list of arguments in parentheses. But exceptions apparently exist: for instance, arithmetic expressions are written in the usual infix way, e. g., as `1+2`. Similarly, some of Prolog's built-in predicates are usually called alike, like the predicate `is/2`$_{\text{ISO}}$ to evaluate an arithmetic expression in `?- X is 1+2`.

Support for the infix notation of `+/2` in arithmetic expressions could be taken as granted, as it is a common language feature among all modern programming languages. However traditionally, statements are written by their instruction or operator first, e. g., in Lisp and assembly languages. In the functional programming language Haskell, functions are usually called using this prefix notation, i. e., the function name followed by its arguments – though unlike in Prolog no parentheses are required. In Haskell, function names consisting of only special characters (like `+` and `++`) are expected two be infix operators; all other names of binary functions can be enclosed by the back quote character `` ` `` to use them in infix notation, e. g., in `3 `mod` 2`.

Though Prolog similarly allows to use graphic symbols like `+` as atoms without the need for additional quote characters, there is no automatism which considers them as infix operators just based on their syntax as in Haskell. Instead, operators have to be explicitly declared in Prolog, using the `op/3`$_{\text{ISO}}$ predicate. It is usually used as a directive in a Prolog program (cf. Section 3.1.2), since the definition of all used Prolog operators is required for the compiler to successfully parse the program they appear in. It has the signature `op(+Precedence,+Type,:Name)`. Multiple operators of the same type and precedence can be defined by a single call of `op/3`$_{\text{ISO}}$ with a list of all names as its last argument. Specifying an operator's precedence as 0 via `op/3`$_{\text{ISO}}$ effectively removes all existing definitions of this operator.

A declaration of an operator via `op/3`$_{\text{ISO}}$ only affects the supported external appearances of the operator in the program's source code. Internally, the standard representation for compound terms, called *functional notation* or *canonical notation*, is still used, i. e., `1+2` is handled in Prolog as it was `+(1,2)`, without any means to distinguish it from the *operator notation* using the operator after the Prolog program was read in. Every compound term can be expressed in functional notation, thus making the syntax without any operator definitions a subset of Prolog in which all programs can be stated non-ambiguous. Throughout this work at hand, we use the phrase *operand* in contrast to *argument* to indicate it is part of a compound term stated in the operator notation, though semantically there is no difference between the operator's notation "operand" and the functional notation's "argument".

Listing 5.1: A simple Prolog rule and its equivalent term in functional notation.

```prolog
p(X, Y) :- X is Y+2*3.
% representation as a compound term in functional notation:
%   :-(p(X,Y),is(X,+(Y,*(2,3))))
```

5.1.1. Precedences in the Parsing of Expressions

In addition to the operator's name, the parsing of a term in functional notation relies on the operator's priority over other terms. It is required to disambiguate expressions in which the structure of a term is not made explicit through the use of parentheses. For instance, the term `1+2*3` with infix operators `+/2` and `*/2` could be parsed either as `+(1,*(2,3))` or `*(+(1,2),3)`. Therefore, every definition of an operator also contains its *precedence*, which is an integer from 1 to 1200. Note that the ISO Prolog standard uses only the broader notion of *priority*, which on the contrary is defined as an integer from 0 to 1201.[22] Though this notion is not as frequently used in literature, we use it similarly throughout the work at hand to distinguish a term's priority from the precedence of the term's principal functor, i. e., the corresponding operator's property defined via $\texttt{op/3}_{\texttt{ISO}}$.

The term's *principal functor* is determined by and equal to the operator with the highest precedence, i. e., the priority of just `1+2` equals the precedence that is defined for `+/1` in the $\texttt{op/3}_{\texttt{ISO}}$ directive. Following the general notation of arithmetic terms, `1+2*3` should be parsed as `+(1,*(2,3))`, thus the precedence of `+/2` is higher than that of `*/2`.

Because Prolog is homoiconic, these considerations can be extended to complete rules. As shown in Listing 5.1, the principal functor of the term that represents the rule is `:-/2`. It should have the highest precedence, followed by `is/2`, `+/2`, and finally `*/3`. All in all, the rule can be represented by the canonical term `:-(p(X,Y),is(X,+(Y,*(2,3))))`.

5.1.2. Infix Operator Associativity

Another ambiguity can be found in terms with multiple occurrences of the same operator, as they have the same precedence. For instance, the term `1+2+3` may represent either `+(1,+(2,3))` or `+(+(1,2),3)`, i. e., the term's principal functor could be the first or the second occurrence of `+/2`. This is determined by the operator's

[22] The priority of variables and atoms as well as of terms written in functional notation or in parentheses is 0. The additional value of 1201 is introduced to describe terms that cannot be built from operators, and could be equally defined as having a priority of infinity (`inf` in Prolog).

type (or *specifier* in the ISO Prolog standard). For infix operators, the type is one of `xfx`, `xfy`, or `yfx`. In these identifiers, `f` denotes the position of the functor name of the infix operator, and `x` and `y` indicate the operator's associativity. In case of `y`, the operand at this position has to be of a priority equal to or less than the operator's, and alternatively in case of `x` it must be of strictly lower priority. For the examples in this section, we assume that for each type an operator with the same name with the same type has been defined, i. e., `xfx/2` denotes the binary operator of type `xfx`.

Following the restrictions to the placeholder `x`, the type `xfx` denotes an operator that is not associative at all – both operands are required to be of strictly less precedence than the operator itself. However, terms can always be explicitly written in parentheses, which gives it zero priority, so `1 xfx (2 xfx 3)` and `(1 xfx 2) xfx 3` are valid terms again.

In contrast, the type `xfy` is right-associative. Only the second operand must be of lower priority; the other can be of the same priority as the principal functor's precedence, effectively allowing to omit the parentheses. The left-associative type `yfx` is the other way round. Thus the terms `1 xfy 2 xfy 3` and `1 yfx 2 yfx 3` are valid Prolog terms, representing the compound terms `xfy(1,xfy(2,3))` and `yfx(yfx(1,2),3)`. There is no type `yfy`, as it would again make the term `1 yfy 2 yfy 3` ambiguous, with the first or second appearance of `yfy` as the term's principal functor.

In the exemplary rule for `p(X,Y)` in Listing 5.1, using the associative type `xfy` would allow that, e. g., `is/2` is of the same precedence as `+/2`, as parentheses are then also not required in the rule's definition. In the ISO Prolog standard, however, these two operators are defined with different precedences, as it allows `is/2` to be of type `xfx` instead of `xfy`, thus preventing the otherwise meaningless term `X is 1 is 2` just by syntactical restrictions. As can be seen, for practical applications the operator's type and precedence have to be chosen carefully, and often depend on each other. A main result of our work in Chapter 9 is the automatic deduction of appropriate operator definitions based only on example sentences.

5.1.3. Prefix and Postfix Operators

Besides the types `xfx`, `xfy`, and `yfx` for binary functors, Prolog also allows the definition of unary prefix and postfix operators. The types are `fx`, `fy`, `xf` and `yf`, indicating by `x` and `y` the same restrictions regarding their operand's precedence as before. For instance, the predicate for negation `\+/1`[C.2] is usually used as a prefix operator, i. e., written as `\+ g` instead of `\+(g)` for a goal g that is expected to be

Listing 5.2: Outputs by `write_canonical/1`$_{\text{ISO}}$, `write/1`$_{\text{ISO}}$, and the toplevel for a given rule.

```
?- T = (p(X, Y) :- X is Y+2*3),                                    TOPLEVEL
   write_canonical(T), nl, write(T).
:-(p(X,Y),is(X,+(Y,*(2,3)))) % functional notation
p(X,Y):-X is Y+2*3            % operator notation where feasible
T = (p(X,Y):-X is Y+2*3) .    % computed answer substitution for query
```

not provable. Here, using the type `fy` over `fx` makes the double application without parentheses in `\+ \+` g a valid term.

Whether it is defined as infix, prefix, or postfix is called the operator's *class*. It is possible to have more than one operator of the same name, as long as they are of different classes. The ISO Prolog standard contains the restriction that there should be no infix and postfix operators with the same name, as it allows a parser to decide immediately the type of an operator without too much look ahead. Nevertheless, several systems like SWI-Prolog and SICStus Prolog lift this restriction, as the operator's class is decidable unambiguously at latest when the term or Prolog program is read in completely. We discuss this language extension as `allow_infix_and_postfix_op`[D.13] in Chapter 10.

5.1.4. Common and Predefined Operators and Predicates

The definition of the prefix operator `\+/1`[C.2] is part of the ISO Prolog standard. The complete list of built-in operators is depicted in Table 5.1. With the exception of `,/2`, they can be overridden and deleted (by precedence zero) per module. To examine the set of operators currently in force, the built-in predicate `current_op/3`$_{\text{ISO}}$ can be used.

The built-in predicate `write_canonical/`$\{1,2\}_{\text{ISO}}$ writes the given term using only the functional notation, i. e., ignoring all operator definitions and enclosing atoms by single quote characters `'` where necessary. The alternative `write/`$\{1,2\}_{\text{ISO}}$ in contrast writes the given term using all appropriate operators. Listing 5.2 shows the different outputs for the rule for `p/2` of Listing 5.1. In the initial query it has to be put in parentheses, as the precedence of `:-/2`$_{\text{ISO}}$ is higher than that of `=/2`$_{\text{ISO}}$ according to the ISO Prolog standard. The first output from `write_canonical/1`$_{\text{ISO}}$ strictly uses the functional notation. The second output, which acknowledges the operator definitions of the ISO Prolog standard, resembles our initial query, with only some spaces being omitted. This is possible, because it is known that an unquoted atom like `:-` consists of only special characters, thus the subsequent `X` concludes the

Table 5.1: Operator table of the ISO Prolog standard.

Precedence	Type	Operators
1200	xfx	`:-` `-->`
1200	fx	`:-` `?-`
1100	xfy	`;`
1050	xfy	`->`
1000	xfy	`,`
900	fy	`\+`
700	xfx	`=` `\=` `==` `\==` `@<` `@=<` `@>` `@>=` `=..` `is` `=:=` `=\=` `<` `=<` `>` `>=`
500	yfx	`+` `-` `/\` `\/`
400	yfx	`*` `/` `//` `rem` `mod` `div`[1] `<<` `>>`
200	xfx	`**`
200	xfy	`^`
200	fy	`+`[1] `-` `\`

[1] Added in Technical Corrigendum 2 of [55].

Table 5.2: Additional operators defined by SWI-Prolog.

Precedence	Type	Operators
1150	fx	`dynamic` `multifile` `discontiguous` `initialization` `thread_local` `thread_initialization` `public` `module_transparent` `meta_predicate` `volatile`
1105	xfy	`|`
1050	xfy	`*->`
990	xfx	`:=`
700	xfx	`=@=` `\=@=` `as` `>:<`
600	xfy	`:`
500	yfx	`xor`
500	fx	`?`
400	yfx	`rdiv`
100	yfx	`.`
1	fx	`$`

graphical symbol and starts a new variable symbol. The output of the toplevel uses the operator notation by default and is therefore the same as the previous line.[23]

In addition to the built-in operators defined in the ISO Prolog standard, all Prolog systems define commonly used operators, whose definitions are either loaded automatically or as part of a module. In Table 5.2, we list the operators that are additionally defined by SWI-Prolog:

- The `fx`-operators of precedence 1150 are usually used as directives in a Prolog program. The predicates `dynamic/1`$_{\text{SWI}}$, `multifile/1`$_{\text{SWI}}$, `discontiguous/1`$_{\text{SWI}}$, and `initialization/1`$_{\text{SWI}}$ declare a predicate property as defined in ISO 7.4.2. Equivalent counterparts for multi-threaded applications [136, Sec. 10] are provided by `thread_local/1`$_{\text{SWI}}$ and `thread_initialization/1`$_{\text{SWI}}$. Predicates, whose definition changes during run-time because of asserting or retracting clauses (cf. Section 3.7), should be declared by the `dynamic/1`$_{\text{SWI}}$ directive beforehand. The predicates `public/1`$_{\text{SWI}}$ and `module_transparent/1`$_{\text{SWI}}$ define additional predicate properties used by SWI-Prolog, or for compatibility with other systems. The directive `meta_predicate/1`$_{\text{SWI}}$ allows to declare meta-predicates as introduced in Section 3.6. Finally, `volatile/1`$_{\text{SWI}}$ declares a predicate to not be saved in the program's saved state [136, Sec. 13.2].

- Technical Corrigendum 2 of [55] defines the bar `|` as a token but not operator. It shall be only an infix operator with precedence greater than or equal to 1001, SWI-Prolog uses 1105. If not defined as an operator, the bar can be used only infix as the separator of head and tail in the list notation of terms (cf. Section 3.3.1).

- SWI-Prolog provides short notations for the *If-then* predicate using the operators `->/2`$_{\text{SWI}}$ and `*->/2`$_{\text{SWI}}$. The latter uses a soft-cut and is known under the name `if/3` in some other Prolog systems.

- The operators `:/2`$_{\text{SWI}}$, `>:</2`$_{\text{SWI}}$, `./2`$_{\text{SWI}}$, and `:=/2`$_{\text{SWI}}$ have been added to SWI-Prolog in version 7 for working with dicts (cf. Section 3.3.5). The first allows the notation of key-value pairs. `>:</2`$_{\text{SWI}}$ provides a modified unification for dicts, while the others are used to access keys and apply functions on dicts.

- The infix operators `=@=/2`$_{\text{SWI}}$ and `\=@=/2`$_{\text{SWI}}$ define the *testing for variant*, which is weaker than equivalence via `==/2`$_{\text{ISO}}$, but stronger than unification via `=/2`$_{\text{ISO}}$.

- The prefix operator `$/1`$_{\text{SWI}}$ can be used in the toplevel to access a variable of a previous query and is mainly intended for debugging purposes and the

[23] The default output format of the toplevel can be changed by the flags `answer_write_options` and `print_write_options`.

interactive work with a program in the toplevel. The operator `as/2`$_{\texttt{SWI}}$ is used by SWI-Prolog's module system. Finally, the operators `xor/2`$_{\texttt{SWI}}$ and `rdiv/2`$_{\texttt{SWI}}$ define functions on Boolean values and rational numbers.

Note that though a functor name is known to be an operator – either built-in by the ISO Prolog standard or in the used Prolog system, or user-defined after an explicit call of `op/3`$_{\texttt{ISO}}$ – it does not mean that an operation is performed when such an operator is encountered. There are many built-in operators that neither implement an arithmetic function nor a corresponding predicate. The definition of an operator alone has no effect on a program's semantics (despite the removal of ambiguities in the parsing of terms). Instead, an operator only enables more expressive ways to rephrase an otherwise equivalent program of functional notation.

5.2. Program Transformations via Term Expansions

With no distinction between code and data, Prolog makes it easy to work on and modify Prolog source code from within the language. As introduced in Section 3.7, facts and rules in an already loaded program can be removed and added at run-time with the help of the built-in predicates `retract/1`$_{\texttt{ISO}}$, `retractall/1`$_{\texttt{ISO}}$, and `assert{a,z}/1`$_{\texttt{ISO}}$. Instead of modifications to Prolog's database at run-time, all major Prolog systems additionally provide a mechanism that adds support for macro expansions, and conditional compilation that is performed already at compile-time. This mechanism is called *term expansion*. For the expansion of bodies of clauses, it is also referred to as *goal expansion*. Its advantage compared to dynamic predicates is clear: complex program transformations can be made once at compile-time instead of first loading a Prolog program and then rewrite the clauses at run-time. As a result, term expansions allow to write concise programs, whose performance is still reasonable, as they do not rely on just-in-time indexing of dynamic clauses, and might profit from static code optimisations that happen only at compile-time.

5.2.1. Implementation and Usage of Term Expansions in SWI-Prolog

Though term expansions are not part of the ISO Prolog standard, most Prolog systems provide a similar interface. In this section, we give a short introduction to the term expansion as performed by SWI-Prolog. It is based on its manual [136, Sec. 4.3.1]. The implementation of the underlying built-in predicates `term_expansion/2`$_{\texttt{SWI}}$ and `goal_expansion/2`$_{\texttt{SWI}}$ as well as the performed order of steps might vary in other Prolog systems. A comparison of the means that

various Prolog systems provide to expand Prolog terms at compile-time is given in [133, 134].

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

1. *Handling of conditional compilation directives.*
   To simplify writing portable code, SWI-Prolog introduced several directives that can be used to enclose code fragments, whose compilation is tied on prerequisites. For instance, with the help of `if/1`$_{\text{SWI}}$ and `endif/0`$_{\text{SWI}}$, it is possible to first check if a goal provided as the argument in `if/1`$_{\text{SWI}}$ is true, and only then compile the code in-between the block built by `if/1`$_{\text{SWI}}$ and `endif/0`$_{\text{SWI}}$:

   ```prolog
   :- if(Goal).
   % platform dependent code
   :- endif.
   ```

   Conditional compilation is usually the method of choice to load different libraries on different Prolog systems or dialects. Similarly, missing predicates which are commonly built-in for other systems can be defined.

2. *Execution of user-defined term expansions.*
   Prolog developers can define term expansions in the predicate `term_expansion(+Term1,-Term2)`. The first argument is a Prolog term as it occurs literally in the source code that is to be compiled. The second argument denotes its replacement. This can be for example a list of clauses. This way it is possible to replace a simple clause by a complex set of facts and rules. Note that definitions of `term_expansion/2`$_{\text{SWI}}$ can be put at any place in a Prolog source code file, though they apply only to clauses following behind. To define new expansions, the full power of Prolog rules can be used. In particular, it is possible to define multiple term expansions with the same first argument `Term1`, as the correct one is picked at compile-time using backtracking, in case the body of the `term_expansion/2`$_{\text{SWI}}$ rule fails. On the other hand, the first successful term expansion is used and cannot be undone. Consequently, possible term expansions should be stated from the most specific to the most general alternative.

3. *Call expansion for definite clause grammars.*
   SWI-Prolog ships with the built-in predicate `dcg_translate_rule/2`$_{\text{SWI}}$. It is called to expand embedded definite clause grammars into plain old Prolog code. We introduce the expansion scheme for DCGs in more detail in Section 6.2.

4. *Execution of goal expansions.*
   As mentioned before, it is possible to define expansions that apply only for goals appearing in the body of rules. This is done with the help of the user-defined predicate `goal_expansion/2`$_{\texttt{SWI}}$. It is similar to `term_expansion/2`$_{\texttt{SWI}}$ of step 2 and is applied to all rule bodies that appear in the program or are produced in one of the previous steps.

These steps are repeated for all newly created terms until a fixpoint is reached. In particular, it is possible to define an expansion that replaces a DCG grammar rule by another one, which later gets translated into normal Prolog code using SWI-Prolog's `dcg_translate_rule/2`$_{\texttt{SWI}}$.

If for a loaded clause no unifying and succeeding definition of `term_expansion/2`$_{\texttt{SWI}}$ and `goal_expansion/2`$_{\texttt{SWI}}$ is found, the term is handled without modifications, i. e., acting as if the all-matching facts `term_expansion(T,T)` and `goal_expansion(T,T)` were defined.

5.2.2. Term Expansions for TAP Test Generation

We use term expansions in all of our projects for the definition of tests. Our *library(tap)* allows to specify tests in Prolog following the *Test Anything Protocol (TAP)*.[24] It defines a standard to output results from test tools independent of the used programming language and system. This interface is supported by a wide range of tools for running, rendering and analysing the test results. At the time of writing this thesis, our *library(tap)* is the only Prolog package to produce TAP-conform text output. The add-on for SWI-Prolog has originally been written by Michael Hendricks and is maintained by the author of this thesis since February 2019.

In general, the assertions in any testing environment have to be specified in a well-defined format, which therefore acts as a domain-specific language for test specifications. Since Prolog is homoiconic, tests can be specified using the same language as the program, i. e., just by Prolog clauses. For instance, in *library(plunit)* – the testing environment for SWI-Prolog and SICStus Prolog – tests are given in the body of `test/`$\{1,2\}_{\texttt{SWI}}$ rules, which again have to be placed in blocks between `begin_tests/1`$_{\texttt{SWI}}$ and `end_tests/1`$_{\texttt{SWI}}$ directives [140, Sec. 8.2].

In *library(tap)*, we do not rely on those special predicates and directives. Instead, tests are specified simply as Prolog clauses, and expanded after loading the module, as shown in the example given in Listing 5.3. Predicates that should be tested or

[24] Test Anything Protocol, https://testanything.org/.

Listing 5.3: Specification of tests with *library(tap)*.

```
1 % possibly load external libraries                                    PROLOG
2 :- use_module(to_be_tested).
3 % define tested or helper predicates
4 equal(A, B) :- A =:= B.
5
6 :- use_module(library(tap)). % all following clauses are expanded
7 'two plus two is four' :- 4 is 2+2.
8 'zero not equal to one'(false) :- equal(0, 1).
9 6 is 3*2.
```

are required to perform the tests can be either loaded from external modules (l. 2), or defined in the same source code file the tests are specified. For instance, the predicate `equal/2` is defined in line 4 and tested with arguments `0` and `1` in line 8.

By term expansion, all clauses following the including of *library(tap)* (l. 6) are extended into tests with the predicate's name as the test name. For small tests (l. 9), the name can be omitted. Then, the test body is used as its name. This is achieved by the following Prolog code snippet, which serves as a small example on how to expand a fact into a rule:

```
1 term_expansion(Fact, (Head :- Fact)) :-                               PROLOG
2   \+ functor(Fact, :-, _),              % Fact is not a rule
3   Fact \== end_of_file,
4   format(atom(Head), "~w", [Fact]),     % Head = '6 is 3*2'
5   tap:register_test(Head). % remember to call the test '6 is 3*2'
```

Given the original clause `6 is 3*2` in line 9 of Listing 5.3, this fact is replaced by a rule with the same head as an atom (created by the de-facto standard predicate `format/3`$_{\text{SWI}}$), and the original clause in its body. The first clause in the body of `term_expansion/2`$_{\text{SWI}}$ (l. 2) ensures that only facts are expanded, i. e., the examined term is not of the functor `:-/2`$_{\text{ISO}}$. The next line is necessary because SWI-Prolog offers the special term `end_of_file` to add clauses at the very end of a file, which should not be tested and replaced by our *library(tap)*. The special term `end_of_file` serves as an anchor and makes it possible to expand an otherwise empty file into a complex Prolog program. The predicate `register_test/1` provided by *library(tap)* later executes the tests by meta-calling the added rule with an head of `Head` using `call/1`$_{\text{ISO}}$.

5.2.3. Preventing Name Conflicts with Built-in Predicates

A common problem when defining internal Prolog DSLs is the potential risk to name collisions with already existing built-in predicates, either provided by the ISO Prolog standard or used Prolog system. As introduced in Section 5.1.4, the definition of operators can be overridden and deleted per module by redefining the operator's type and precedence (of possibly zero in case of deletion). However, this affects only the possible ways to represent terms, but does not impact clauses describing the corresponding predicate. Predicates already defined in the `system` module (cf. Section 3.8) are automatically loaded in all modules, resulting in conflicts in case the internal DSL's term representation uses identical functors.

For instance, in our example in Listing 5.3, the fact `6 is 3*2` describes a test case for the `is/2` predicate. The term expansions provided by *library(tap)* transform this fact into a new rule of the predicate `'6 is 3*2'/0`. Without the described term expansion, the Prolog compiler throws an error once the `is/2` fact is encountered, as the ISO Prolog standard predicate is already defined in the `system` module, namely to calculate arithmetic expressions. SWI-Prolog returns the following message when loading the source code of Listing 5.3 without *library(tap)*:

```
ERROR: No permission to modify static procedure 'is/2'          TOPLEVEL
```

There are two approaches to work around this name collision: explicitly redefining the system predicate, or wrapping definitions of and calls for this predicate by term and goal expansions. Both techniques allow to reuse predicate names that are otherwise reserved for built-in predicates, thus broadening the range of languages that can be described as an internal Prolog DSL.

Redefine System Predicate. The ISO Prolog standard avoids conflicts of names of user-defined and built-in predicates by simply disallowing clauses about predicates that are part of the standard (ISO A.1.4). Only with modules as introduced in Section 3.8, it is possible to define two predicates with the same functor but different meanings. But since predicates and operators of the `system` module are always available, the internal DSL's predicates ought to be explicitly qualified via the operator `:/2`$_{\texttt{SWI}}$ to avoid conflicts, resulting in more verbose clauses like `6 test:is 3*2`.

SWI-Prolog allows to give alternative clauses for any system predicate after its explicit declaration via the `redefine_system_predicate/1`$_{\texttt{SWI}}$ directive. In Listing 5.4, we give an example for redefining the binary built-in predicate `is/2`$_{\texttt{ISO}}$ in the `user` module. The `redefine_system_predicate/1`$_{\texttt{SWI}}$ directive was originally introduced to facilitate the compatibility between different Prolog systems. It allows new clauses

Listing 5.4: Redefining the `is/2`$_{\text{ISO}}$ predicate in the `user` module with an additional indirection to the original predicate in the `system` module.

```
1 :- redefine_system_predicate(is(_,_)).                    PROLOG
2 prolog is nice.
3
4 % manually add bypass for built-in predicate as very last clause
5 is(A,B) :- system:is(A,B).
6 % or alternatively state term expansion anywhere
7 term_expansion(end_of_file, (is(A,B) :- system:is(A,B))).
```

of built-in predicates in any module, including the `user` module, which makes the alternative predicate definition available in all modules. The system's original definition is no longer available in the default `user` namespace, thus often breaking compatibility with other modules. This can be overcome by adding a clause which maps the predicate's new definition to the original from the `system` module. It can be given manually as the very last clause of this predicate, as presented in line 5 of Listing 5.4. Alternatively, this clause can be added anywhere in the source code as a term expansion for the special `end_of_file` term, as defined in line 7.

Wrap Predicate via Term Expansions. Term expansions are also the means of choice to avoid name conflicts with built-in predicates in the first place. Instead of redefining the predicate of the `system` module, the predicates formed by sentences in the internal Prolog DSL are translated into predicates with unique functors. For instance, the user-defined predicate `is/2`, which otherwise conflicts with in the `user` module, can be expanded into facts of the functor `dsl_is/2`:

```
1 term_expansion(A is B, dsl_is(A, B)).                     PROLOG
2 goal_expansion(A is B, (dsl_is(A, B) ; is(A, B))).
```

Propositions stated in the internal Prolog DSL, for instance `prolog is nice`, are translated into facts of the predicate `dsl_is/2` (l. 1). If a goal calls `is/2`, clauses of this newly created predicate `dsl_is/2` are applied first, and the system predicate `is/2`$_{\text{ISO}}$ alternatively (l. 2).

Note that both approaches – redefining the system predicate via the `redefine_system_predicate/1`$_{\text{SWI}}$ directive, and wrapping the user-defined predicate with the help of term expansions – expect the predicates used in the internal DSL and the built-in predicates to be of the same instantiation mode. For the presented `is/2` wrappers, we expect only queries with the original mode `(-,+)` of `is/2`$_{\text{ISO}}$. If additional modes are required, the predicate of the `system` module has

to be wrapped accordingly, namely to either catch possible instantiation errors, or to ensure it is called only with correctly bound values via `var/1`$_{\texttt{ISO}}$ and `nonvar/1`$_{\texttt{ISO}}$ (cf. Section 3.2.1) in the first place.

5.3. Program Execution with Meta-Interpreters

In Sections 3.7 and 5.2 we introduced two techniques to manipulate Prolog programs: *dynamic predicates* allow to modify the Prolog database at run-time, while *term and goal expansions* provide macro expansions that are performed at compile-time. Both approaches effectively translate the knowledge given in an internal Prolog DSL back to plain old Prolog clauses, thus relying just on Prolog's underlying SLD resolution evaluation method and the provided built-in predicates.

Another approach to work with the knowledge specified in the form of an internal Prolog DSL is by interpretation. With Prolog's built-in capabilities for reflection and term inspection as introduced in Sections 3.5 and 3.6, facts and rules can be accessed and processed as Prolog terms. In addition, meta-predicates like `call/`$n_{\texttt{ISO}}$ allow to apply dynamically created goals. Together, this makes Prolog well-suited for writing interpreters for Prolog programs in Prolog itself, thus creating a *meta-interpreter* (MI).

Meta-interpreters allow to interpret, execute, and analyse the integrated data format. This lifts the internal Prolog DSL from a language that just describes knowledge in the form of Prolog predicates which are consultable from the Prolog toplevel, to an executable program performing the operations defined by the meta-interpreter – this way, meta-interpreters bring the internal DSL to life.

5.3.1. Vanilla Meta-Interpreter

The most basic meta-interpreter for Prolog is the *vanilla meta-interpreter* [114, pp. 323f]. It does not add any others features than just applying a given goal based on the known facts and rules using Prolog's normal execution method of SLD resolution. To execute the vanilla meta-interpreter, we define the predicate `mi/1`, which has the same functionality as the built-in predicate `call/1`$_{\texttt{ISO}}$. As shown in Listing 5.5, only few lines of code are necessary to cover all possible cases that form a goal. If the given goal is `true/0`$_{\texttt{ISO}}$, the predicate `mi/1` succeeds (l. 2). Goals which are compound terms of `,/2`$_{\texttt{ISO}}$ representing a logical conjunction (l. 3), or of `;/2`$_{\texttt{ISO}}$ representing a disjunction (l. 4), succeed if the first and/or second argument is satisfiable using the meta-interpreter. Unifications succeed or fail just like in Prolog, i. e., for compound

Listing 5.5: Definition of the vanilla meta-interpreter for Prolog programs.

```prolog
1 %% mi(:Goal)
2 mi(true).                          % for facts
3 mi((A , B)) :- mi(A), mi(B).       % conjunction
4 mi(A ; B) :- mi(A) ; mi(B).        % disjunction
5 mi(A = B) :- A = B.                % unification
6 mi(A) :-
7   A \= true, A \= (_ , _), A \= (_ ; _), A \= (_ = _),
8   clause(A, B),
9   mi(B).
```

terms of functor `=/2`$_{\text{ISO}}$ the corresponding unification is applied (l. 5). For all other cases, a clause whose head unifies with the current goal is determined using the built-in meta-predicate `clause/2`$_{\text{ISO}}$. In case of a fact, the clause's body `B` is bound to `true/0`$_{\text{ISO}}$. Otherwise, the meta-interpreter continues to evaluate the clause's body, as the goal `A` should succeed only if the clause's body `B` can be proven. If a goal cannot be satisfied by the current clause, `clause/2`$_{\text{ISO}}$ provides alternative clauses using backtracking, in the order they appear in the source code. Besides the logical disjunction via `;/2`$_{\text{ISO}}$, it is the interpreter's only source of backtracking.

5.3.2. Adaptions to Handle DSLs and Nondeterminism

The vanilla meta-interpreter has no additional practical benefit compared to the built-in `call/1`$_{\text{ISO}}$ predicate. However, it serves as a starting point to write a meta-interpreter that is specific for the given internal Prolog DSL. In addition, the meta-interpreter allows to adjust the standard search strategy of the SLD resolution, and treat its sources of nondeterminism (cf. Section 2.4.2) differently:

- The default selection function s_l takes the left-most literal in a conjunction of subgoals. This strategy is reflected in line 3 of Listing 5.5. Alternatively, the right-most literal can be taken, i. e., the meta-interpreter evaluates the second argument first. In case of a random selection function s, the compound binary term of `,/2`$_{\text{ISO}}$ has to be flattened into a list of subgoals first, e. g., using the predicate `term_functors_list/3`$_{\text{[C.3]}}$.

- The second source of nondeterminism in the SLD resolution – the order in which the possible clauses are applied –, can be addressed in line 8 of Listing 5.5. Instead of relying on Prolog's default order of their appearance in the source code, all clauses with a unifying head A could be collected first, and then arbitrarily sorted.

– By default, the SLD resolution traverses the search tree in a depth-first manner. Changing this requires larger modifications in the meta-interpreter, as the current depth of the search tree has to be stored in an additional argument, resulting in the predicate `mi/2`. Then, alternatives – either from logical disjunctives (l. 4) or multiple clauses (l. 8) – are only applied if the current depth limit is not exceeded.

In almost all applications, the meta-interpreter for an internal Prolog DSL can use the nondeterminism handling as provided by the SLD resolution. Then, adapting the meta-interpreter to the DSL's requirements leads to more clauses of `mi/2` that describe alternative compound terms as their arguments.

5.4. Declarative If-then Rules for Expert Knowledge

The operator notation of terms allows to write Prolog programs that closely resemble a natural language. This way, all graphical symbols in facts and rules can be eliminated and replaced by more descriptive operator names. Only the full stop `.` is required to mark the end of a Prolog clause and thus cannot be eliminated – but this happily complies with the normal notation of ending written sentences. By setting appropriate operator precedences and types, propositions can be stated without the need for parentheses.

In this section, we use the merits of Prolog to define a declarative internal DSL for representing expert knowledge in the form of rules. It serves as an example for a simple domain-specific, Prolog-based data format to represent knowledge as rules and is revived again in Chapter 6 to illustrate the definition of external DSLs. A first version of this internal DSL and its formal grammar is presented in our work in [110] and its extended journal version in [109]. It has been applied to represent expert knowledge in the field of change management in organisational psychology. The conclusions and applications for the research field of change management are described in [126], which justifies this approach to use an internal Prolog DSL for rapid prototyping and incremental knowledge bases of real-world applications.

5.4.1. Definition as an Internal DSL

The expert knowledge was obtained by surveys in the form of organisational rules, and then stated in a simple, textual, and logic-based format. To express a single rule, we choose to follow the general form:

`if` *Condition* `then` *Consequence*. PROLOG DSL

Though semantically the rules are similar to Prolog's Horn clauses, they are represented in a natural language syntax, avoiding Prolog's rule separator symbol `:-/2`$_{\text{ISO}}$, and allowing the more natural statement order which specifies the condition first. After the keywords `if` and `then`, a *condition* and a *consequence*, respectively, is expected. Both are so-called junctions of *findings*. A finding has the basic form *Feature* `=` *Value*. Here, the infix operator `=/2` denotes "is" or "equals". Besides equality, additional comparison operators are supported, like the symbols `>/2` for "greater than" and `</2` for "smaller than". Since the findings are Prolog terms, the feature and value are usually strings or atoms. Therefore, they either start with a lowercase letter or have to be enclosed in single or double quotation marks. For simple Boolean expressions that denote the existence or absence of a given feature, the values `yes` and `no` can be used, as they represent the Prolog constants `yes/0` and `no/0`. Numerical values can be stated verbatim in the Prolog DSL. They are of particular practical importance, as they later allow to define rules that are based on weights, probabilities, and their calculations.

In case of multiple findings in the rule's condition and consequence, they can be linked to formulas by the connectives `and` and `or`, which are implemented as binary operators in the internal DSL. Here, as for Prolog, conjunction binds stronger than disjunction. Classical negation can be expressed using the keywords `not` and `neg`, which are defined as unary prefix operators. The two different operators have been defined only for convenience, and can be used interchangeably.

The text-based DSL is conveniently defined in Prolog by a collection of suitable user-defined operators and their precedences, as listed in Table 5.3. The bindings of the connectives `and` and `or` are as described, as `and/2` has a lower precedence than `or/2`. For both, we use infix operators of type `yfx`, as it allows to state multiple junctions without the need for parentheses. All other operator types use `x`-specifiers to avoid nesting of otherwise meaningless terms already at the syntactic level of the DSL. The precedences are chosen to comply with the values from the built-in operators `is/2`$_{\text{ISO}}$ (700) and `\+/1`$_{\text{ISO}}$ (900) – though our prefix operators for classical negation, `not/1` and `neg/1`, use the type `fx` instead, as unlike `\+ \+` *g* the term `not not` *g* should not be supported in our DSL.

As a simple example sentence, we express that clothes get wet if (i) the weather is rainy and one has no umbrella, or (ii) there is a thunderstorm. This statement can be modelled as an instance of this internal Prolog DSL as follows:

```
if weather = rainy and umbrella = no                    Prolog Dsl
   or weather = thunderstorm
then clothes = wet.
```

Table 5.3: Operators that define the internal Prolog DSL for if-then rules.

Precedence	Type	Operators
1100	xfx	then
1000	fx	if
900	fx	not neg
850	yfx	or
800	yfx	and
700	xfx	is[1] are
200	fx	a an the no

[1] Built-in operator defined in the ISO Prolog standard.

In this listing and the following, we arbitrarily indent the terms only for the sake of improved readability. Because Prolog is not whitespace-sensitive, the program's meaning does not rely on the actual indentation, i. e., we could similarly write the complete Prolog program in a single line. Since the subjects `weather`, `umbrella`, and `clothes` begin with a lowercase letter and do not contain whitespaces, they form Prolog atoms without further modifications, and can be stated verbatim. Similarly, the adjectives `rainy` and `wet` can be expressed without the need to be enclosed in quotation marks. In the given sentence, they form operands to the built-in operator `=/2`$_{\text{ISO}}$. The complex term that describes the proposition as a sentence is built from the user-defined operators `and/2`, `or/2`, `if/1`, and `then/2` from Table 5.3.

With the definition of the additional operators, the internal DSL can be further extended to resemble natural language even more. Firstly, the more descriptive built-in infix operator `is/2`$_{\text{ISO}}$ can be used instead of the equal symbol. It just denotes the term `is(·,·)` – as before with `=/2`$_{\text{ISO}}$, which is usually the predicate symbol for unification, it does not mean that an arithmetic evaluation is performed when the operator `is/2`$_{\text{ISO}}$ is encountered somewhere in the term. For plural forms, we equally define an infix operator `are/2`. Finally, we provide the determiners `a`, `an`, `the`, and `no` as prefix operators that bind the most strongly.

The complete Prolog program with the operators defined in Table 5.3 is given in Appendix B.1. It allows to specify the example sentence as the small Prolog program of Listing 5.6:

Listing 5.6: Example if-then rule in the internal DSL.

Prolog Dsl

```
if the weather is rainy and there is no umbrella
   or the weather is a thunderstorm
then the clothes are wet.
```

The single clause is equal to the following program, where all terms are written in the canonical notation instead of relying on the operators:

```
then(if(or(and(is(the(weather), rainy),                    PROLOG
               is(there, no(umbrella))),
           is(the(weather), a(thunderstorm)))),
     are(the(clothes), wet)).
```

This program is consultable in Prolog without further modifications. Knowledge given in the described form of if-then rules can be asked for by goals of `then/2`, for instance:

```
?- if Condition then Consequence.                          TOPLEVEL
Condition = (the weather is rainy and there is no umbrella or the
   weather is a thunderstorm),
Consequence = (the clothes are wet) .
```

5.4.2. Binary Expression Tree

Figure 5.1 illustrates the nested term from Listing 5.6 as a binary tree. The inner nodes depict the operators as defined in Table 5.3. Binary operators like `then/2` have two child nodes. The dashed edge on the left describes the binary operator's first operand, and the solid edge on the right the second operand. Prefix operators (types `fx` and `fy`) are depicted as if they were binary operators without a first argument, i. e., with only a single solid edge to the right. Though we do not use it in our example, postfix operators equally would have only a dashed edge to the left. The binary tree's leaves are formed by the real-world entities the proposition is about, e. g., `weather`, `umbrella`, and `rainy`.

This encoding of terms in operator notation forms a binary expression tree. With in-order traversal (left, node, right), the original sentence can be built again. For instance, the highlighted part in Figure 5.1 represents the Prolog term `the weather is rainy`. In this binary expression tree, the original expressions are built by recursively producing the left part first, then printing out the operator depicted in the node, and finally producing the right expression recursively again. Since `the/1` is defined as a prefix operator of type `fx`, the first part reads as `the weather` in in-order traversal.

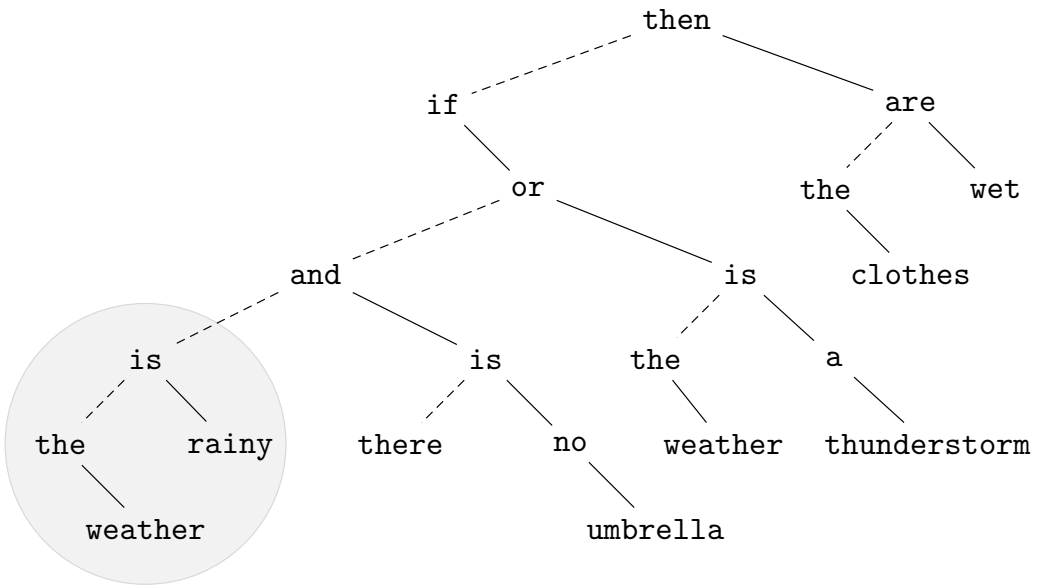

Figure 5.1: Binary tree representation of an example sentence in the internal DSL. The highlighted part represents the Prolog term `the weather is rainy` when read in infix-order.

5.4.3. Expanding If-then Rules to Plain Old Prolog Clauses

Term expansions as presented in Section 5.2 allow to transform the if-then rules into plain old Prolog rules. Instead of the descriptive prefix and infix operators `if/1` and `then/2`, the built-in rule notation via `:-/2`$_{\texttt{ISO}}$ can be used. As a consequence, the expert knowledge is not just stated as consultable facts in the form of `then/2` predicates, but instead can be used to infer knowledge from a given set of facts and rules. Additionally, the native Prolog representation allows to combine the expert knowledge with the full power Prolog provides, while the representation as an internal Prolog DSL can still be used.

We assume that expert knowledge in this internal DSL is either stated in the form of the if-then rules we introduced before, or as facts like `the weather is rainy`. Not different to Prolog, conjunctions of facts can be simply stated using multiple clauses; disjunctions of facts (e. g., `the weather is rainy or the weather is sunny`) are not supported, as they do not represent Horn clauses. Then, it is known that `and/2` and `or/2` appear only in the body of if-then rules, i. e., they can be expanded by goal expansion (step 4 in Section 5.2.1). The if-then rules in the form of `then/2` facts on the other hand are replaced by term expansions (step 2). Listing 5.7 shows the appropriate user-defined clauses of `term_expansion/2`$_{\texttt{SWI}}$ and `goal_expansion/2`$_{\texttt{SWI}}$. For the sake of simplicity in this example, we assume the name conflict with the built-in predicate `is/2`$_{\texttt{ISO}}$ is resolved by just declaring so in the `redefine_system_predicate/1`$_{\texttt{SWI}}$ directive as introduced in Section 5.2.3.

Listing 5.7: Definition of term and goal expansions for if-then rules.

```prolog
goal_expansion(Cond1 and Cond2, (Cond1 , Cond2)).
goal_expansion(Cond1 or Cond2, (Cond1 ; Cond2)).
term_expansion(
  if Condition then Consequence,
  (Consequence :- Condition) ).
```

The infix operator `and/2` is replaced by the built-in conjunction `,/2`$_{\text{ISO}}$ (l. 1), the operator `or/2` by calls of the built-in disjunction `;/2`$_{\text{ISO}}$. Because the steps for the term expansion are repeated until a fixpoint is reached and our operators `and/2` and `or/2` are of the same type and relative precedences as the built-in `,/2`$_{\text{ISO}}$ and `;/2`$_{\text{ISO}}$, this also correctly replaces nested terms like `A or B and C` by `A ; B , C`, preserving the intended logical reading and operator precedences of the nested terms. As a result, the program given in the internal Prolog DSL is translated into plain old Prolog clauses and logical connectives as expected. The code listing via `listing/1`$_{\text{SWI}}$ (cf. Section 3.5) reveals the translated code:

```prolog
?- listing(are).
the clothes are wet :-
  ( the weather is rainy,
    there is no umbrella
  ; the weather is a thunderstorm ).
```

In addition to the single if-then rule from Listing 5.6, further rules and facts can be stated using this internal Prolog DSL, for instance:

```prolog
if the umbrella is broken then there is no umbrella.
the weather is rainy.
the umbrella is broken.
```

This knowledge again gets replaced by plain old Prolog facts and rules by the term expansions we introduced in Listing 5.7. As a result, Prolog's normal resolution technique can be used to infer knowledge. For instance, it is possible to retrieve everything that is known just by calling the query `?- X is Y ; X are Y` (to get both singular and plural forms) in the toplevel:

```prolog
?- X is Y ; X are Y.
X = the weather, Y = rainy ;   % the weather is rainy
X = the umbrella, Y = broken ; % the umbrella is broken
X = there, Y = no umbrella ;   % there is no umbrella
X = the clothes, Y = wet .     % the clothes are wet
```

Listing 5.8: Definition of a meta-interpreter for if-then rules.

```prolog
mi(true).                       % for facts
mi(A and B) :- mi(A), mi(B). % conjunction
mi(A or B) :- mi(A) ; mi(B). % disjunction
mi(A = B) :- A = B.          % unification
mi(A) :-
  A \= true, A \= (_ = _),
  ( clause(A, B)     % there is a Prolog clause A :- B
  ; (if B then A) ), % or there is a fact then(if(B),A)
  mi(B).
```

Here, the proposition `the clothes are wet` is inferred from the knowledge base stated in the internal Prolog DSL by SLD resolution. The complete program with all operator definitions, facts, and if-then rules is given in Appendix B.1. To infer the knowledge by Prolog's SLD resolution as presented, the term expansions given in Listing 5.7 have to be added at any point before sentences in the form of the internal DSL are formulated.

5.4.4. Meta-Interpretation

As an alternative to the translation of the internal DSL for if-then rules to plain old Prolog code by term expansion, the provided propositions can be interpreted instead. Only minor changes to the vanilla meta-interpreter we introduced in Section 5.3 are required. Listing 5.8 shows the meta-interpreter that processes if-then rules. Instead of Prolog's built-in operators `,/2`$_{\text{ISO}}$ and `;/2`$_{\text{ISO}}$ for logical conjunction and disjunction, it defines similar interpretations for the user-defined infix operators `and/2` and `or/2` (ll. 2–3). In addition to the vanilla meta-interpreter's usage of `?- clause(A,B)` (which searches for Prolog clauses of the form `A :- B`), the meta-interpreter for if-then rules also uses rules of the form `if B then A`. The resulting meta-interpreter allows to query the knowledge base given in the form of the internal Prolog DSL, without the need to expand it into plain old Prolog clauses first.

5.5. From DSLs to Controlled Natural Languages

The operators we presented in the previous section and in Table 5.3 can be further extended to support a wider range of language constructs. For instance, the binary operator `than/2` allows to specify comparisons as Prolog terms like `X is smaller than 3`, representing the term `is(X,than(smaller,3))`, with a variable `X`, the comparator `smaller` as an atom, and the number `3`. To its full extent,

this results in an expressive grammar, based only on the Prolog syntax and its existing compilers. Consequently, all tools originally defined to assist with the development of Prolog programs (e. g., for debugging, syntax highlighting) can be used when new propositions are expressed in this internal DSL.

The sentences that can be stated this way are a subset of a natural language for the DSL's particular field of applications. In this regard, the internal Prolog DSL serves as a simple *controlled natural language* (CNL). In our work [108], we compare these two approaches – DSLs and CNLs – for representing knowledge in intelligent systems in a declarative and natural way. With applications from rapid prototyping to large intelligent systems, Prolog has been proven useful in both: for implementing internal DSLs as introduced in this chapter, but also for the definition of a large subset of standard English, as for instance with Attempto Controlled English (ACE) [42, 43]. Both approaches allow domain experts to express complex sets of rules without having to learn a formal computer language first.

In principle, there is a huge overlap between DSLs and CNLs. In particular for query languages like SQL, it cannot be determined if they are defined as a DSL or CNL just from the outside, as their syntax often resembles questions verbalised in natural language. On the other hand, languages that specify grammars, e. g., regular expressions, could be easily stated as a subset of a natural language, but are instead defined as a DSL, with postfix operators like `+/1` to represent the otherwise more descriptive notation of "at least once". Further DSLs for rule bases in medicine have been created in [106, 107]. In our work [83], we rely on the Prolog-based Attempto Controlled English instead of Prolog DSLs to interactively express and process expert knowledge using smart voice-controlled devices.

5.6. EBNF as an Internal DSL for Context-free Grammars

The syntax of many formal languages is specified in the form of a context-free grammar, as it allows to implement efficient parsers that, for a given string, determine whether and how it can be generated from the grammar. The widely used *LR(k) parsers* (bottom-up parser, reading input from left to right, producing the rightmost derivation in reverse) and *LL(k) parsers* (top-down, left-to-right, leftmost derivation) analyse deterministic context-free grammars in linear time [63], given an upper bound $k \geq 1$ of tokens to look-ahead.

The extended Backus–Naur form (EBNF) is a popular notation to formally describe context-free grammars using production rules. It is used by most ISO standards that define programming, markup, or query languages, including EBNF on its own [58].

The list of EBNF rules consists of nonterminals and terminals, which are called *symbols*. Symbols are typically alphanumeric characters, punctuation marks, and similar, specified in quotation marks.

Each EBNF rule has three parts: a *left-hand side* of just a single nonterminal, a *right-hand side* consisting of nonterminals and symbols, and the = symbol which separates the two sides and reads as "is defined as". The elements of the right-hand side either describe an ordered sequence (denoted by commas ",") or alternative choices (denoted by vertical bars |, with a smaller precedence than the ordered sequence). Repetitions are enclosed by curly brackets { ... }, optional nonterminals by square brackets [...], and comments by brackets of the form (* ... *). The semicolon character ";" marks the end of rule, as unlike BNF it might span across multiple lines.

As an example, we consider in Figure 5.2 an extract of the ISO Prolog standard (Iso 6) that specifies the syntax of a variable, which we informally introduced in Section 3.1.1. In Prolog, most tokens are allowed to be preceded by an arbitrary number of whitespace characters and comments, which are summed up in the optional nonterminal *layout text sequence*. A *variable token* is either the anonymous variable given by the underscore character, or a named variable which has to start with an underscore character or an uppercase letter. For instance, `_` is the anonymous variable, and `_a` and `A` are named variables. The comments in the EBNF refer to the sections of the ISO Prolog standard where the other referred nonterminals are defined.

The EBNF given in Figure 5.2 requires only minor modifications to be valid Prolog code, as `;/2`, `=/2`, and `'|'/2` can be defined as infix operators with sensible types and precedences. By manually adjusting to the following requirements, the EBNF already forms an internal Prolog DSL:

- Nonterminals must be valid Prolog atoms, so included whitespaces have to be replaced, e. g., by underscore characters, or the nonterminal identifier has to be enclosed in single quotation marks.
- The full stop `.` has to end the very last rule.
- Comments are written as `/* ... */` instead of `(* ... *)`.

The term `(* 6.4.1 *)` taken by itself is valid Prolog syntax if `*/1` is defined as both a prefix and postfix operator, since every Prolog term is allowed to be additionally bracketed, thus representing the term `*(*(6.4.1))`, independent from the chosen operator precedences. However, this requires the inner operand to be a valid Prolog term. Only by chance this is the case for `6.4.1`, because it represents the binary

EBNF

variable = [*layout text sequence* (∗ 6.4.1 ∗)],
variable token (∗ 6.4.3 ∗) ;
variable token = *anonymous variable* (∗ 6.4.3 ∗)
| *named variable* (∗ 6.4.3 ∗) ;
anonymous variable = *variable indicator char* (∗ 6.4.3 ∗) ;
named variable = *variable indicator char* (∗ 6.4.3 ∗),
alphanumeric char (∗ 6.5.2 ∗),
{ *alphanumeric char* (∗ 6.5.2 ∗) }
| *capital letter char* (∗ 6.5.2 ∗),
{ *alphanumeric char* (∗ 6.5.2 ∗) } ;
variable indicator char = *underscore char* (∗ 6.5.2 ∗) ;
underscore char = "_" ;
capital letter char = "A" | "B" | "C" | ... ;
alphanumeric char = ...

Figure 5.2: EBNF grammar rules for a variable in Prolog.

compound term `.(6.4,1)` for `./2` being defined as an infix operator, and the floating-point number `6.4` as its first argument. Nevertheless, the EBNF comment is not necessarily a valid Prolog term, e. g., in case of arbitrary text, which cannot be stated verbatim. In addition, consecutive Prolog terms are allowed only as operands of compound terms. Consequently, a comma or some other operator is required in front of the comment, since the bracketed term `(* 6.4.1 *)` has to be an argument.

The notations of EBNF for optional nonterminals and repetitions thereof can be used identically in the internal Prolog DSL. The first simply constitutes a list with a single element using the square bracket notation. The second, EBNF's repetitions, are enclosed in curly brackets { ... }. This also forms a valid Prolog term according to the ISO Prolog standard (Iso 6.3.6). The so-called *curly bracketed term* `{t}` is equal to the term `{}(t)` with a principal functor of `{}/1`$_{\text{ISO}}$.

The complete definition of all required operators to use EBNF as an internal Prolog DSL is given in Appendix B.2, together with the consultable example Prolog program that defines the context-free grammar to parse a variable token. The EBNF grammar given in Figure 5.2 is then represented by the following Prolog program written in canonical notation with the classical operator `./2` for the list construction. Its term's principal functor is `;/2`. The indentation and line breaks serve the readability:

PROLOG

```
;( =( variable,
     ','( .( layout_text_sequence, [] ), variable_token )),
```

```
;( =( variable_token,
      '|'( anonymous_variable, named_variable )),
   ;( =( anonymous_variable, variable_indicator_char ),
      ;( =( named_variable,
            '|'( ','( variable_indicator_char,
                      ','( alphanumeric_char,
                           {}( alphanumeric_char ))),
                 ','( capital_letter_char, /*...*/ ))),
         ;( /*...*/ ))))).
```

This term, which represents the context-free grammar given in the internal Prolog DSL, again serves as a starting point for either transforming it via term expansions, for meta-interpretation, or as-is for queries in the Prolog database. For instance, we provide in Appendix B.3 a term expansion which translates this internal DSL for EBNF into plain old definite clause grammars, which are supported by and shipped with all major Prolog systems.

6

Integration of External DSLs with Quasi-Quotations and DCGs

I once thought Prolog was poorly standardised, but now I know better. SQL is very poorly standardised.

— Jan Wielemaker[25]

The definition of a domain-specific language internally in Prolog has various advantages. Firstly, all development tools that are available for Prolog can be similarly used for the integrated language: IDEs allow syntax highlighting, code linters allow to check the adherence to coding style guides, and static code analysis might provide performance improvements. In addition, the internal DSL can be easily integrating into existing Prolog environment, gaining the full advantage of Prolog's built-in SLD resolution to infer knowledge, and its large set of community-run libraries. Data given in the form of the internal DSL creates an incremental knowledge base, and is particularly suitable for rapid prototyping.

These advantages all originate from the fact that the internal DSL solely relies on the Prolog parser, which is an integral component of any Prolog system. This architectural strength of defining a DSL internally is at the same time its greatest weakness: it requires the integrated language to be a valid subset of Prolog. And though Prolog's syntax is flexible and without any keywords – even built-in predicates and operators can be deleted and redefined –, adhering closely to the ISO Prolog standard comes with some unchangeable rules the DSL has to comply with. Among others, the two most important are:

- Words have to start with a lowercase letter in order to build atoms instead of variables. Otherwise, as well as in case they contain whitespaces, they have to

[25] Quote from the documentation of SWI-Prolog's ODBC interface, https://github.com/SWI-Prolog/packages-odbc/blob/bad664/README. Jan Wielemaker is a Dutch computer scientist and professor at the University of Amsterdam. He is the original author and today's main developer of SWI-Prolog.

be put between single, double, or back quotation marks, representing atoms, strings, or lists.

– All sentences have to end with the full stop. In particular, `.` has to be the very last symbol in the document that holds the DSL.

Since documents holding data given in the internal DSL are processed by the normal Prolog parser, the error handling is based only on Prolog's syntax and not adjusted to the requirements of the domain-specific language. On the one hand, in case of invalid Prolog programs this results in error messages that are neither descriptive enough nor related to the actual DSL. While on the other hand, the Prolog parser might accept sentences in the internal DSL which are perfectly valid Prolog programs, but are not intended by the integrated DSL. For instance, potential errors like starting an entity with an uppercase letter will not yield any warning, since it is recognised as a variable which is syntactically allowed in the same positions as atoms by the Prolog parser. Consequently, the sentence `the Weather is _rainy` admittedly is a valid Prolog term for `the/1` being defined as a prefix operator and `is/2` as an infix operator. The symbols `Weather` and `_rainy` though are Prolog variable names according to the ISO Prolog standard, so the stated proposition is much more general than originally intended. These syntactic pitfalls of an internal Prolog DSL might be particularly hard to catch for users – the experts of the application domain –, which are not very experienced with Prolog.

These disadvantages can be addressed by using a specialised parser for the DSL, instead of relying on the Prolog parser that comes with every Prolog system. Though this approach to define a domain-specific language externally lowers the barrier to entry for users not familiar with Prolog, the implementation overhead is significant, as a fully featured parser requires in-depth knowledge about compilers, state machines, language theory, and grammars. Besides, additional steps to make the external DSL executable are still required and the same as we introduced for internal domain-specific languages in Sections 5.2 and 5.3: after the parsing process, the syntax tree representing the external language has to be either translated into corresponding Prolog clauses via term expansion, or traversed through by a user-defined interpreter.

Nevertheless, Prolog has a long history of defining parsers, as it has been applied in the research field of natural language processing for decades. In 1978, Colmerauer introduced *Metamorphosis Grammars* [22], a first framework based on first-order logic to parse French. Its rewriting rule mechanism led to the development of *definite clause grammars* in 1980 [96]. This formalism to define grammars is similar to EBNF, but based on Prolog's execution model and the underlying SLD resolution. It

therefore comes with logic variables and backtracking, resulting in context-sensitive grammars.

In this chapter, we adapt and extend the existing considerations and tools to work with DCGs for the integration and definition of any external DSL. It extends typical constructs and problems known from natural language processing to formal and computer languages. Section 6.1 presents the two ways of connecting an external DSL with Prolog: the document holding the domain-specific language can be read in and parsed from an external file, or put verbatim next to plain old Prolog code using the recently introduced quasi-quotations. The parsing of an external DSL requires a language specification in the form of a grammar. In Section 6.2, we introduce Prolog's de-facto standard for specifying grammar rules, definite clause grammars (DCGs). It extends EBNF, which we defined as an internal Prolog DSL in Chapter 5, to context-sensitive grammars. Section 6.3 continues our considerations from Section 5.4 on if-then rules and presents their definition as an external DSL. The chapter concludes in Section 6.4 with a discussion on how to integrate the application layer query language GraphQL as an external DSL and use it in combination with Prolog predicates.

6.1. Embed External DSLs in SWI-Prolog

Connecting the domain-specific language with Prolog is straightforward in case of an internal DSL. Since it is just valid Prolog code, it can be put right next to and worked with just like any other Prolog source code. Only if the operators or predicates with identical functors are already defined in the existing codebase, the definition of the internal DSL has to be taken into a separate module to avoid name conflicts.

The integration of an external DSL into a Prolog codebase on the other hand requires additional steps, as code of the DSL and the host language cannot be intertwined. The classical approach is to split the code into files of the integrated domain-specific language on the one hand and the host language on the other. Since version 6.3.17, SWI-Prolog additionally provides a special syntax to enclose any external language. With so-called quasi-quotations, the foreign code can be put again right next to native Prolog source code into a single file.

6.1.1. Processing Content from the Outside-World

To use a document which is given in the form of an internal DSL but in a separate file, all of Prolog's built-in capabilities to load code can be used:

– The ISO Prolog standard predicate `read/`$\{1,2\}_{\texttt{ISO}}$ reads a Prolog term given in the internal DSL from an input stream and unifies it with the variable given as its last argument. The stream can be created by opening a file, or just using the standard input `stdin`. The latter is particularly useful to prompt the user to enter a term given in the internal DSL.

– The de-facto standard predicate `consult/1`$_{\texttt{SWI}}$ reads from a given filename and parses its content as Prolog source code. Traditionally, it may be abbreviated by just typing a number of filenames in a list, e. g., `?- [dsl]` to load the Prolog file `dsl.pl`.

– If the DSL is given in a separate module, it can be loaded via the predicate `use_module/`$\{1,2\}_{\texttt{SWI}}$ (cf. Section 3.8).

All three means to load source code given in an internal DSL rely on the Prolog parser shipped with the used Prolog system and are therefore not applicable for external DSLs. For such, there are alternative predicates which are similar but expect an additional parameter to the grammar that should instead be used to process the contained source code. Only for the predicate `use_module/`$\{1,2\}_{\texttt{SWI}}$ there is no equivalent for external DSLs, because modules are expected to always be valid Prolog programs.

The central predicate to process content from the outside-world is the predicate `phrase/3`$_{\texttt{ISO}}$, which is defined in the proposed *Part III* [57] of the ISO Prolog standard. It works on an input given as a list, since Prolog's traditional data representation for strings is based on lists, as introduced in Section 3.3.4. Then, the goal `?- phrase(:Grammar, ?List, ?Rest)` is true if the elements in `List` can be processed by the grammar rule `Grammar`, leaving the list's remainder `Rest` – i. e., `List-Rest` describes a difference list (cf. Section 3.3.3). The predicate `phrase/2`$_{\texttt{ISO}}$ is a short form for the special case of `Rest = []`, i. e., the specified grammar rule is required to process all list elements. The formalism to describe grammars like the referenced `Grammar` as well as possible implementations of the predicates `phrase/`$\{2,3\}_{\texttt{ISO}}$ will be presented in the next Section 6.2.

The predicate `phrase/3`$_{\texttt{ISO}}$ expects that the external document's content has already been loaded into a list, the actual file reading is left to a preprocessing step. In this regard, `phrase/3`$_{\texttt{ISO}}$ is similar to the predicate `read/`$\{1,2\}_{\texttt{ISO}}$. SWI-Prolog additionally provides the predicates `phrase_from_stream/2`$_{\texttt{SWI}}$ and `phrase_from_file/`$\{2,3\}_{\texttt{SWI}}$. They are part of the built-in module `pure_input` from *library(pio)*, and directly process the content from a stream or file. Though both predicates internally make use of `phrase/3`$_{\texttt{ISO}}$, they use attributed variables (which we introduce in more detail in Section 10.1.3) to lazily read the input from an external source into the processed

list on demand with the help of open difference lists. These custom-designed predicates should therefore be preferred over `phrase/3`$_{\texttt{ISO}}$ when reading in an external DSL from a separate file.

6.1.2. Code-Inlining with Quasi-Quotations

Originally, Prolog has poor support for long text fragments, which are often needed for the integration of external DSLs. Though there are several mechanisms to specify strings that span about multiple lines, they all require the user to adjust the original multi-line string by appropriate escape characters. The ISO Prolog standard prohibits newlines that are not escaped to appear in quoted material, though it used to be common practice before and is still allowed by some systems, like SWI-Prolog. Following the ISO Prolog standard, newlines in strings have to be escaped by a trailing single backslash `\`. In addition, most Prolog systems also support the character escape sequence `\c`. Both differ only in the system's handling of following whitespaces. Since the character escape sequence `\c` skips all graphical characters up to but not including the next non-layout character, it allows to arbitrarily indent lines in the integrated external DSL, without changing the corresponding string representation in Prolog.

To support multi-line strings natively and to simplify the embedding of external DSLs without the need of manually adding escape symbols, multiple extensions to Prolog's syntax have been discussed in [130]. Inspired by constructs in other programming languages, for instance Haskell and JavaScript (where it is called *tagged template strings*, cf. Section 4.3.3), *quasi-quotations* have been added to SWI-Prolog in 2013 [137]. Since then, they have been used to embed several well-known external languages like HTML, SQL, and the semantic query language SPARQL. As of now, SWI-Prolog is the only Prolog system with support for quasi-quotations.

The basic form of a quasi-quotation is as follows, where `Tag` is a callable predicate with an arity of 4, and `Content` is a string, which holds the document given in the form of an external DSL:[26]

```
{| Tag || Content |}                                              SWI-Prolog
```

The document given in the external DSL, `Content`, is an arbitrary sequence of characters from the system's alphabet (cf. Section 3.1). If the document contains the quasi-quotation's closing character sequence `|}`, it must be escaped according

[26] In the following and throughout our thesis, code examples might include both quasi-quotations and SWI-Prolog's dicts (cf. Section 3.3.5), which might confuse the reader. Although their visual appearance is similar, they can easily be differentiated by their start token: quasi-quotations are opened by `{|`, whereas dicts begin with `Tag{`.

to the rules of the target language. All other characters, in particular newlines, can be stated in the quasi-quotation without further modifications.

As an example, the HTML fragment for a paragraph holding the text `Hello, Name!` can be embedded in SWI-Prolog using a quasi-quotation. The tag can be arbitrarily chosen, we use `html(doc)`:

```
{|html(doc)||<p>Hello, Name!</p>|}
```
SWI-Prolog

Quasi-quotations are allowed in a Prolog program at any position where a term is expected, i. e., they form a fifth option in the inductive definition of terms we introduced in Section 3.1.1. All embedded documents given in quasi-quotations are translated at compile-time into plain old Prolog terms with the help of term expansion and a grammar defined by the user. Given a quasi-quotation with the tag `Tag = TagName(SyntaxArgs...)`, the compiler therefore calls the corresponding user-defined predicate `TagName/4` as follows:

```
TagName(+ContentHandle,+SyntaxArgs,+Vars,-Result) :- % ...
  % ... user-defined processing of the content to create Result
  %     usually based on a grammar, for instance DCGs
```
Prolog

In our example of a HTML paragraph, the term has the tag `html(doc)` and invokes the following call to the user-defined predicate `html/4`:

```
html('<p>Hello, Name!</p>', [doc], ['Name' = 'Alice'], Result)
```

The quasi-quotation's tag `Tag` can be any term with the functor `TagName/TagArity`. Its arguments are passed to the predicate `TagName/4` as the list `SyntaxArgs`, which guarantees that the predicate `TagName` is always of arity 4, independent from the `Tag`'s original arity `TagArity`.[27] The argument `Vars` provides access to variables of the quasi-quotation's outside-world context. It can be used to fill in placeholders when parsing the contained document and creating a native Prolog term representation. Though in our example we use the variable symbol `Name` with a value of `'Alice'`, it does not necessarily stand for the `Name` that appears in the HTML paragraph – as it is given in an arbitrary DSL, Prolog's syntax rule may not apply to the embedded document.

The argument `ContentHandle` is an opaque term that carries the content of the quasi-quoted text and position information about the source code and its layout. For the sake of simplicity, instead of SWI-Prolog's internal reference `ContentHandle`

[27] This can be equally expressed with the help of the *univ* operator (cf. Section 3.6): `?- Tag =.. [TagName|SyntaxArgs], length(SyntaxArgs, TagArity)`

we directly use the embedded source code fragment `Content` as an atom in our code examples. The content of the embedded document is usually passed to `with_quasi_quotation_input/3`$_{\texttt{SWI}}$ or `phrase_from_quasi_quotation/2`$_{\texttt{SWI}}$. While the first predicate creates a stream, `phrase_from_quasi_quotation/2`$_{\texttt{SWI}}$ parses the enclosed string according to a given grammar, just like the original `phrase/2`$_{\texttt{ISO}}$.

6.2. Definite Clause Grammars

The embedding of external DSLs in Prolog is closely connected with the built-in predicates `phrase/`$\{2,3\}_{\texttt{ISO}}$. Independent from how the DSL is integrated into the Prolog codebase – either in a separate file, or directly interwoven with the Prolog code using quasi-quotations –, the connected DSL has to be parsed and translated into a Prolog term. This term then represents the external document, either as a concrete or abstract syntax tree, or as its interpretation. For this purpose, the predicate `phrase(:Grammar,?List,?Rest)` applies a grammar on the difference list `List-Rest`. In general, the predicate can be called in various instantiation modes. For instance, `(:,+,-)` processes a given list and calculates all of its remainders according to the given grammar via backtracking. If on the other hand `List` is a free variable instead, i. e., `phrase/3`$_{\texttt{ISO}}$ is called as `(:,-,-)`, the same grammar can be used to generate all allowed documents of the external DSL. As a result, a language described in a formalism that can be consumed by `phrase/`$\{2,3\}_{\texttt{ISO}}$ can often be used to parse a given document and translate it into a corresponding Prolog term, or vice versa to serialise the document back based on a given term representation.

The first parameter of `phrase/`$\{2,3\}_{\texttt{ISO}}$ is a callable term (cf. Section 3.3) which denotes the grammar rule that should be used for parsing and serialisation. It is expected to be a Prolog goal of functor p/n that can be called with two additional arguments provided by `phrase/3`$_{\texttt{ISO}}$, i. e., there is a corresponding Prolog predicate $p/(n+2)$. For instance, the goal `?- phrase(float(N), [-,3,.,1,4], X)` invokes a call for `float(N, [-,3,.,1,4], X)`. Given a sensible implementation of the predicate `float/3` that allows both integer and floating-point numbers, the initial goal computes the three answer substitutions `I = -3, X = [.,1,4]`, `I = -3.1, X = [4]`, and `I = -3.14, X = []`.

With respect to its first argument, the predicate `phrase/`$\{2,3\}_{\texttt{ISO}}$ is just a meta-predicate which wraps calls for the provided goal. Though the corresponding predicate can be ordinarily defined using Prolog clauses, *definite clause grammars* (DCGs) provide a short notation to express those grammar rules without having to always specify the predicate's two additional arguments. DCGs are not yet part of the ISO Prolog standard, but are considered to be officially included in the future as

Part III [57] of the standard. Nevertheless, as of today, DCGs are supported by all major Prolog systems.

6.2.1. Syntax

A definite clause grammar is a set of *grammar rules*. Syntactically, they are similar to Prolog's rules, except they use `-->/2`$_{\text{ISO}}$ instead of `:-/2`$_{\text{ISO}}$:

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

Nonterminals and Terminals. In the basic form of a DCG, `Head` is a predicate with an arbitrary number of arguments, representing a *nonterminal* in the grammar. To distinguish the notation for functors specified in grammar rules from those of normal Prolog predicates, a nonterminal's name and arity are separated by two slashes instead of one. A nonterminal p with n arguments is denoted by the functor $p//n$.

The grammar rule's right-hand side is a sequence of one or more nonterminals and terminals. A *terminal* is a consumed (or produced, respectively) list item, and written in square brackets like normal Prolog lists. For instance, the rule's right-hand side element `[a,b,c]` expects the three atoms `a`, `b` and `c` in the input list. It has the same effect as the conjunction `[a], [b], [c]`. Though DCGs are often used to process strings, i. e., a list of characters, the list items can be of any type, including free variables and compound terms. It is also possible to specify a string enclosed in double quote characters as a terminal, as it represents a list of characters in case of the appropriate setting of Prolog's `double_quotes` flag (cf. Section 3.3.4).

The grammar rule's meaning is similar to those known from EBNF. The nonterminal `Head` represents the sequence of body items $\texttt{B}_1$`, ...,` $\texttt{B}_n$ described in the rule's right-hand side. Unlike Prolog's short notation for facts, there is no short form like a prefix `-->/1` to specify empty DCG bodies. A rule which describes `e//0` as an empty string is specified as one of the following:

```
1 e --> [].
2 e --> "".
```
PROLOG

Control Structures. Besides the conjunction `,/2`$_{\text{ISO}}$, the Prolog control predicates for disjunction `;/2`$_{\text{ISO}}$, if-then-else (with infix operators `->/2`$_{\text{ISO}}$ and `;/2`$_{\text{ISO}}$), and the prefix operator `\+/1`$_{\text{ISO}}$ for negation can be used to express relationships between grammar body items. The operator `|/2`$_{\text{SWI}}$ is defined for compatibility with EBNF to

denote alternatives. All these control predicates behave as in regular Prolog clauses, and can be used together with parentheses to pilot the processing of the grammar.

Prolog code can be embedded by using curly brackets `{...}`. In addition, the cut can be used as usual and is predefined as the nonterminal `!//0`$_{\texttt{ISO}}$. Although it often improves the parsing performance, it can result in wrong or insufficient answers when called the other way round for serialisation. The usage of the cut nonterminal `!//0`$_{\texttt{ISO}}$ should therefore be used only with care.

Semicontext. Apart from having only a single nonterminal as the grammar rule's left-hand side, DCGs also allow the definition of rules in the form `Head, P --> B`$_1$`, ..., B`$_n$, i. e., with a conjunction of a nonterminal `Head` and a list `P` in the rule's head. The terminal `P` is called *semicontext* or *pushback list*.

6.2.2. Procedural Semantics

The grammar rules describe how to rewrite the rule's head into its body items. Following Prolog's underlying SLD resolution mechanism, the rules are applied in their order of appearance. A rule is *applied* if the head's nonterminal unifies with the current nonterminal and all body items can be applied. The body items are consumed from left to right. With Prolog's backtracking mechanism, multiple rules for a single nonterminal might be tested.

When applied, a nonterminal processes the list and leaves everything as remainder that is not processed by the rule's right-hand side. In between, every body item operates on the result of the previous one. The embedded Prolog code, which is given in curly brackets, is executed at the specified position. If the grammar rule contains a pushback list, it is prefixed to the remaining terminal list after successfully evaluating the grammar's body.

As an example, Listing 6.1 shows the DCG that describes a palindrome, i. e., a list of items which reads the same backward as forward. The nonterminal `elem//1` takes a single element `X` from the given list. Since we restrict ourselves to palindromes built only from the atoms `a`, `b`, and `c`, it is tested in the embedded Prolog snippet via `member/2`$_{\texttt{SWI}}$. Without this additional check, the nonterminal `palindrome//0` would succeed for a list of any type whose elements appear the same when reading forwards and backwards.

An empty list as well as a list with only a single element are known to be palindromes (ll. 2–3). The general case of longer lists can be defined using the recursive grammar rule of line 4: it is a palindrome, if the list without the first and last element

Listing 6.1: DCG to describe a palindrome of the characters `a`, `b`, and `c`.

```prolog
elem(X)    --> [X], { member(X, [a,b,c]) }.
palindrome --> [].
palindrome --> elem(_).
palindrome --> elem(X), palindrome, elem(X).
```

is a palindrome, and these elements are the same. This is ensured by the grammar rule of `elem//1`, which binds its first argument to the read element, and thus makes it available for comparison via unification in the body of the `palindrome//0` rule.

The argument of `elem//1` provides access to the rule's context, and is thus lifting the DCG of Listing 6.1 to a *context-sensitive* grammar. In contrast, a DCG with only nonterminals without arguments as the rule's left-hand side represents a *context-free* grammar. Though palindromes can be described using a context-free grammar by stating all possible elements in the recursive rule explicitly (i. e., `palindrome --> [a], palindrome, [a]`, and similar for the atoms `b` and `c`, and all additionally allowed terminals), the DCG's support for logic variables makes it possible to use a single rule instead.

As an alternative to additional arguments to the nonterminals, the semicontext notation of DCGs can also be used to provide access to the rule's context. It is a common approach when modelling a context-sensitive grammar rule to push back a Prolog term to the remaining list that describes the current context, which is then consumed by the next possible nonterminal first. This allows to pass around a Prolog data structure on all applied rules. For instance, we could move the argument of `elem//1` to the pushback list on the rule's left-hand side:

```prolog
elem, [X] --> [X], { member(X, [a,b,c]) }.
```

Then, `elem//0` performs the look-ahead precondition that the next list element is one of `a`, `b`, or `c`, as the read element is pushed back to the processed list without modification. Therefore, the difference list processed by the DCG remains unchanged.

A grammar rule `Head --> Body` without an explicit semicontext is equivalent to the rule `Head, [] --> Body` with the empty pushback list `[]` – since it is empty, no elements are put back to the processed list. If on the other hand the pushback list is different from the list of consumed elements, this contradicts the assumptions made to a difference list. For instance, the following clauses are semantically equivalent:

```prolog
% process the atom a , push back atom b
change, [b] --> [a].
change([a|X], [b|X]).
```

However, though the two arguments `[a|X]` and `[b|X]` each describe a difference list on their own (with `[a|X]-X` and `[b|X]-X`), they do not form a difference list together.

6.2.3. Execution via Meta-Interpreter

Definite clause grammars on their own essentially constitute an internal Prolog DSL: with their well-defined syntax shaped around the `-->` symbol, they are a subset of Prolog with the specialised focus on the application area of parsers and serialisers. Like any other internal DSL, the language's semantics can be implemented either by transforming the terms of `-->/2`$_{\text{ISO}}$ into plain old Prolog clauses, or by the definition of a meta-interpreter. In this section, we first focus on the latter approach. In most Prolog systems, DCGs are translated via term expansions instead. The standard expansion scheme is presented in the following Section 6.2.4.

As introduced before, DCGs are usually applied by using the meta-predicate `phrase/`$\{2,3\}_{\text{ISO}}$, for example in `?- phrase(palindrome, P)` or `?- phrase(palindrome, [a,b,a], Rest)`. It therefore serves as a good target predicate to implement the meta-interpreter. The short form `phrase/2`$_{\text{ISO}}$ can be easily mapped to `phrase/3`$_{\text{ISO}}$:

```prolog
phrase(H, A) :- phrase(H, A, []).
```

PROLOG

It might seem desirable to use the more explicit notation `A-Z` as a single term instead of the two separate arguments in `?- phrase(H, A, Z)`. Though the usage of the term with infix operator `-/2`$_{\text{ISO}}$ does not result in a worse program performance thanks to SWI-Prolog's deep indexing [136, Sec. 2.18.1], it is slightly misleading when DCG's semicontext notation is used. This is because grammar rules with a non-empty pushback list do not describe a proper difference list `A-Z`, as `Z` could contain (the pushed back) elements which are not necessarily part of the list `A`.

Listing 6.2 shows the implementation of the `phrase/3`$_{\text{ISO}}$ meta-interpreter for non-terminals and control structures. The first two clauses describe the application of grammar rules, the others handle control structures that are allowed in the rule's bodies on the right-hand side:

- **Line 1:** Given the nonterminal `H`, it is searched for a corresponding DCG rule with a unifying head, whose body `B` is applied.
- **Line 2:** If alternatively there is a rule written in semicontext notation, the rule's body is applied similarly, with the pushback list `P` prepended to the intermediate result `C`, constituting the list `Z`. For `P` being the empty list `[]`,

Listing 6.2: Meta-interpretation of nonterminals and control structures in DCGs.

```
1 phrase(H, A, Z) :- (H --> B), phrase(B, A, Z).                    PROLOG
2 phrase(H, A, Z) :- (H, P --> B), phrase(B, A, C), append(C, P, Z).
3 phrase((B1 , B2), A, Z) :- phrase(B1, A, D) , phrase(B2, D, Z).
4 phrase((B1 ; B2), A, Z) :- phrase(B1, A, Z) ; phrase(B2, A, Z).
5 phrase(\+ H, A, A) :- \+ phrase(H, A, _).
6 phrase({ P }, A, A) :- call(P).
7 phrase(!, A, A) :- !.
8 phrase([], A, A).
9 phrase([T|Rest], [T|A], Z) :- phrase(Rest, A, Z).
```

`?- append(C, P, Z)` results in `C=Z`, i. e., the special case of an empty pushback list is semantically equivalent to line 1.

– **Lines 3 and 4** denote the logical conjunction and disjunction of two body elements, which are processed either consecutively (with the chaining variable `D` for the intermediate result) or alternatively.

– **Line 5:** Many DCG implementations allow to specify negated subgoals in the body. The rule succeeds if the subgoal cannot be applied for the given list. The initial list `A` remains unchanged, as no list element is consumed or produced.

– **Line 6:** Embedded Prolog source code, which is enclosed in curly brackets, gets called. Note that this allows normal backtracking in the Prolog goal `P`.

– **Line 7:** Backtracking can be governed by using Prolog's cut `!/0`$_{\text{ISO}}$ within curly brackets. For convenience, it is usually also defined as a nonterminal `!//0`, so the cut can be stated verbatim in the grammar rule's body.

– Finally, the **lines 8 and 9** describe the handling of nonterminals, which are given in lists. In case of an empty list on the right-hand side of a grammar rule, nothing is read or produced. Consequently, the processed list remains unchanged. Otherwise, the processed list has to start with the head element `T`, and the remainder is applied recursively.

The complete Prolog program with the implementation of `phrase/3` as a meta-interpreter together with an example DCG is given in Appendix B.4.

6.2.4. Standard Term Expansion Scheme

In most Prolog systems, DCGs are translated into plain old Prolog clauses at compile-time via term expansions. In this case, `phrase/3`$_{\text{ISO}}$ does not serve as a

Listing 6.3: Term expansion for DCGs. The complete code is given in Appendix B.5.

```
1 term_expansion(X1 --> Y1, X2 :- Y2) :-                    PROLOG
2   ( X1 = (L, P), append(P, Z, Out)
3   ; X1 = L, Out = Z),
4   term_args_attached(L, [In, Out], X2),                   [C.4]
5   translate_body(Y1, Y2, In, Z).                          [B.5]
```

Listing 6.4: Listing of the generated Prolog clauses for the DCG that describes a palindrome from Listing 6.1.

```
?- listing([elem, palindrome]).                             TOPLEVEL
elem(X, [X|Z], Z) :- true, member(X, [a,b,c]).
palindrome(A, A).
palindrome(A, Z) :- elem(_, A, Z).
palindrome(A, Z) :- elem(X, A, B), palindrome(B, C), elem(X, C, Z).
```

meta-interpreter but a meta-predicate, which calls the predicate $p/(n+2)$ for a given grammar rule $p//n$.

Listing 6.3 presents the term expansion for terms of the form `X1 --> Y1`, which get replaced by a Prolog clause (i. e., a term with functor `:-/2`$_{\text{ISO}}$) `X2 :- Y2`. Lines 2 and 3 handle the two cases for the grammar rule's left-hand side: it is either a pair of a nonterminal `L` and the pushback list `P`, or otherwise only `L`. For an empty pushback list `P`, the two forms are semantically equivalent, as consequently `Out` unifies with `Z` in both lines 2 and 3. A nonterminal $p//n$ is expanded into the predicate $p/(n+2)$ by adding the two additional arguments `In` and `Out` representing the proper difference list `In-Out` (l. 4). It uses the predicate `term_args_attached/3`, which is defined in Appendix C.4. The goal `?- term_args_attached(A,L,B)` creates the compound term `B` by adding the arguments given in the list `L` as new last arguments to the compound term `A`.

This term expansion covers only lines 1–2 from Listing 6.2. The other cases of the meta-interpreter are handled by the referenced predicate `translate_body/4`, which translates the grammar rule's right-hand side `Y1` into the corresponding Prolog rule body `Y2`. It requires the same definition by structural cases as before, but is to some extent more complicated, because the two arguments representing the difference list are added to all body elements, and new chaining variables have to be introduced accordingly. The full definition of the predicate `translate_body/4` is given in Appendix B.5.

In Listing 6.4, we show the listing of the predicates `elem/3` and `palindrome/1` as printed from the toplevel, after the term expansions have been performed. The use

of term expansions has two major advantages compared to the application of DCGs by a meta-interpreter. Firstly, the creation of lists with preceding elements is performed only once at compile-time. As discussed in Section 3.4, the complexity of `append/3` is linear to the length of the first argument, and so is the complexity of the processing of terminals in the meta-interpreter version of `phrase/3`. The generated clauses for the Prolog predicates `elem/3` and `palindrome/2` completely lack calls of `append/3` or list deconstructions. The second advantage originates from the improved possibilities of clause indexing. By creating a new clause for each alternative in the grammar, the translated predicates can be indexed by their arguments, resulting in a greater performance for parsing and serialisation. In the first case, the list prefixes of terminals provide faster access to applicable clauses, and in the latter, the additional arguments of context-sensitive nonterminals can be indexed.

Similar to our term expansion of Listing 6.3, SWI-Prolog translates DCGs into plain old Prolog clauses at compile-time, as one of the four compilation steps we introduced in Section 5.2.1. Internally, the built-in predicate `dcg_translate_rule/2`$_{\texttt{SWI}}$ is used to translate a single grammar rule. It performs additional optimisations to the created Prolog clause. For instance, the tautology `true/0`$_{\texttt{ISO}}$ in the definition of `elem/3` is removed. In our implementation of the term expansions as presented in Listing 6.4, this subgoal is created because each element in a grammar rule's right-hand side is translated into a corresponding Prolog goal. Consuming terminals, however, only performs unifications of the difference list in the clause's head, thus the clause's body is simply `true/0`$_{\texttt{ISO}}$. For simple grammar rules that just consume or produce symbols, SWI-Prolog's `dcg_translate_rule/2`$_{\texttt{SWI}}$ therefore returns just a single fact, putting all information about the processed difference list in the clause's head. This again leads to a faster execution because of SWI-Prolog's clause indexing capabilities.

6.2.5. From EBNF to DCGs

In its simplest form to just specify context-free grammars, DCGs are very similar to EBNF, which we introduced in Section 5.6 and discussed its implementation as an internal Prolog DSL. Like EBNF, a DCG is a list of grammar rules, each consisting of a nonterminal on the left-hand side, and the rule's body on the other. Instead of =, the infix operator `-->/2`$_{\texttt{ISO}}$ is used in between. Alternatives on the right-hand side are denoted by the Prolog operator `;/2`$_{\texttt{ISO}}$ instead of |.

Compared to the context-free grammars of EBNF, DCGs provide three major extensions:

– *Arguments on the left-hand side.* In contrast to EBNF, the nonterminal on the left-hand side of a DCG is allowed to have an arbitrary number of arguments of any type. Since it is common in Prolog to use the same variables for input and output, these additional arguments can be used to both deliver and receive information about the grammar rule's application context.

– *Complex control structures on the right-hand side.* Besides the conjunction `,/2`$_{\text{ISO}}$ and the disjunction `;/2`$_{\text{ISO}}$, all other Prolog control structures can be used in the grammar rule's right-hand side. In addition, any Prolog code can be embedded by using curly brackets `{...}`.

– *Pushback arguments.* DCGs allow the definition of rules in the form `Head, P -->` $\texttt{B}_1$`, ...,` $\texttt{B}_n$, with `P` being a list of terminals that are prepended to the parsed list after successfully evaluating the grammar's body.

With term expansions, EBNF grammar rules stated in the form of an internal Prolog DSL can be translated into normal DCG notation at compile-time. For instance, the single EBNF rule *underscore_char* = "_" gets replaced by the following Prolog fact:

```
term_expansion(A = B, A --> B).
```
PROLOG

With similar term expansions for "|" (alternatives), ";" (rule endings), "?" (optional elements), and "∗" (sequences), the internal DSL is translated into ordinary DCGs, which are supported by all major Prolog systems. As a result, grammars of formal languages provided as EBNF can be directly embedded into and used in Prolog, resulting in efficient and executable Prolog parsers.

6.3. Declarative If-then Rules as an External DSL

In Section 5.4, we introduced the DSL of declarative if-then rules to represent expert knowledge with a natural language flavour. By the definition of appropriate operators, the DSL can be used as an internal DSL, thus relying only on the parser for Prolog programs that comes with every Prolog system. However, every internal Prolog DSL can also be implemented externally using an appropriate grammar. In Listing 6.5, we define a DCG that parses if-then rules of the same format. It serves as a motivational example for an external DSL that is processed using definite clause grammars. Parsing natural language with Prolog has been subject of research for a long time. Nevertheless, the observations made in this chapter can be easily adopted to the integration of computer languages as well as of formal languages. They can

Listing 6.5: DCG to parse and serialise if-then rules.

```
1  fact        --> conjunction, ".".                                    PROLOG
2  rule        --> "if", #, formula, #, "then", #, conjunction, ".".
3  formula     --> conjunction ; disjunction.
4  conjunction --> finding ; finding, #, "and", #, formula.
5  disjunction --> finding, #, "or", #, formula.
6  finding     --> feature, #, equal, #, value.
7  equal       --> "=" ; "is" ; "are".
8  feature     --> "there" ; noun_phrase.
9  value       --> "rainy" ; "broken" ; "wet" ; noun_phrase.
10 noun_phrase --> determiner, #, noun.
11 determiner  --> "a" ; "an" ; "the" ; "no".
12 noun        --> "weather" ; "thunderstorm" ; "umbrella" ; "clothes".
13 #           --> " ".
```

be connected with Prolog as an external DSL with the same means as the simple grammar for English if-then rules of our motivational example.

6.3.1. Definition as an External DSL

The DCG of Listing 6.5 defines the two nonterminals `fact//0` and `rule//0` (ll. 1–2), which serve as the grammar's entry point and are intended for the use with $\texttt{phrase/}\{2,3\}_{\texttt{ISO}}$. Nevertheless, it is possible to use all other nonterminals in the same way. For instance, the goal `?- phrase(noun_phrase, NP)` generates all allowed noun phrases, like `a weather`, and so on.

With the Prolog flag `double_quotes` set to `chars` (cf. Section 3.3.4), nonterminals in a grammar rule's right-hand side can be stated as a string instead of a list. The nonterminal `#//0` succeeds for a whitespace character and separates two consecutive terminals. `a`, `an`, `the` and `no` are possible terminals for the nonterminal `determiner//0`, `weather` and others are possible terminals for the nonterminal `noun//0`. In the internal Prolog DSL, the nouns and values (`value//0`) represent the leaves in the binary expression tree. Being operands, they can be any valid Prolog atom and are not required to be specified in advance in the definition of the internal DSL. Similarly, the DCG could instead consume any non-empty string up to the next whitespace character, using the following grammar rule:

```
noun --> [X], { X \= " " }.                                             PROLOG
noun --> [X], { X \= " " }, noun.
```

In the DCG of Listing 6.5, we explicitly list all allowed nonterminals to be able to use the same grammar to both parse if-then rules and to generate all allowed yet

meaningful sentences. Without, the goal `?- phrase(noun, N)` will not result in a binding for a free variable `N`.

6.3.2. Comparison of the Two Approaches

Using DCGs for parsing external DSLs offers a huge freedom in the design and features of the integrated language, because with Prolog's powerful, built-in technique for parsing and serialisation it is possible to process any string input. With its support for additional parameters to nonterminals, and pushback lists on the grammar rule's left-hand side, combined with possible embeddings of Prolog source code snippets specified in curly brackets, context-sensitive grammars can be easily defined without a thorough understanding of automata theory.

The grammar formalism can help to clarify the syntax of the external DSL for people who are not experts in logic programming or Prolog – particularly, because DCGs with terminals written as strings are easy to understand even if details like the underlying evaluation mechanism, or the handling of embedded Prolog code are unknown. The DCG can be extended by fallback rules as the very last alternative to return meaningful error messages in case of invalid sentences. In addition, the external domain-specific language can be defined to be more relaxed: facts and rules do not necessarily have to end with a full stop `.`, and could be separated by newlines; strings starting with an uppercase letter or containing whitespaces do not have to be encapsulated by quotes.

Though the internal definition of a Prolog DSL does not require the implementation of a fully-featured parser, sentences (i. e., Prolog terms) accepted by the Prolog parser not necessarily conform to the originally intended DSL definition. For instance, grammatical constraints can be ensured only at run-time as part of the meta-interpreter. In contrast, in case of an external DSL, they can be directly encoded in the DCG. To ensure that the determiner `a` is allowed only for nouns starting with a consonant, and `an` only for nouns starting with a vowel, the grammar rule for `determiner//0` can be adjusted as presented in Listing 6.6.

For better readability, we split the alternatives for `determiner//0` into three grammar rules, which is equivalent to the logical disjunction via $;/2_{\texttt{ISO}}$. The first line reads as "`a` is a valid determiner if it is followed by a consonant". Since the following word of `determiner//0` is originally consumed by the `noun//0` nonterminal, the nonterminals `vowel//0` and `consonant//0` are implemented as look-aheads: the consumed whitespace and character `Char` are pushed back to the processed list using DCG's semicontext notation. The check for a vowel is performed by the embedded, user-defined Prolog predicate `is_vowel/1` (l. 6).

Listing 6.6: Extending `determiner//0` by grammatical constraints to use `an` only for nouns starting with vowels, and to use `a` otherwise.

```
determiner              --> "a", consonant.                          PROLOG
determiner              --> "an", vowel.
determiner              --> "the" ; "no".
vowel, [' ', Char]      --> [' ', Char], { is_vowel(Char) }.
consonant, [' ', Char] --> [' ', Char], { \+ is_vowel(Char) }.
is_vowel(Char) :- member(Char, [a,e,i,o,u]).
```

6.4. GraphQL for Deductive Databases

In October 2015, Facebook released *GraphQL*,[28] an open-source application layer query language, which has been internally developed since 2012. It provides a unified interface between the client and the server for data fetching and manipulation. Using GraphQL's type system, it is possible to specify data handling of various sources and to combine, e. g., relational with NoSQL databases. In contrast to the well-established architectural pattern REST, GraphQL provides a single API endpoint and supports flexible yet only previously defined queries over linked data. Already when publicly announced in 2015, most of Facebook's applications made use of GraphQL as their primary data-fetching mechanism, resulting in hundreds of billions of GraphQL API calls a day [15].

Although now used several years in production by Facebook and others, the GraphQL specification is still under active development. The most recent working draft specification is of January 2022,[29] and available at https://spec.graphql.org/. However, most of the changes since its initial publication in October 2015 have been clarifications in the wording as well as formal descriptions of the underlying mechanisms. For instance, GraphQL's *type system definition language* (often referred to as its "SDL") has been formally described in the form of an EBNF in 2018. It is used to describe the capabilities of a GraphQL server.

A detailed introduction to GraphQL and the SDL's grammatical and semantic rules is given in [47]. For a short overview with a focus on comparing it to the well-established REST data layer, we refer to our work [84]. This paper describes our implementation of a GraphQL server in SWI-Prolog. In the section at hand, we focus only on a single part of this framework: the integration of the GraphQL type system and query syntax as an external domain-specific language in Prolog.

[28] GraphQL, "A query language for your API", https://graphql.org/.
[29] As of January 2022.

Listing 6.7: Example GraphQL query and corresponding JSON result.

GraphQL

```
query getAlice {
  person(name: "Alice") {
    name,
    years: age,
    books(favourite: true) {
      # is implicitly a list
      title,
      authors {
        name
      }
    }
  }
}
```

Json

```
{ "data": {
  "person": {
    "name": "Alice",
    "years": 31,
    "books": [
      {
        "title": "Moby-Dick",
        "authors": [{
          "name": "H. Melville"
        }]
      }
    ]
  } } }
```

6.4.1. Example Query and Result

In most applications, complex, structured data is requested from the API. To provide a single entity with all its relations from a single endpoint, a GraphQL query is structured hierarchically. Its structure represents the data that is expected to be returned. In general, each level of a GraphQL query corresponds to a particular type. They can be nested and also recursive. The result document is a set of entities with their relations specified in the type system. This clarifies the name *GraphQL*, as these entities and relations can be thought of as a graph.

As a motivating example, we consider a GraphQL server to access and manipulate data of persons along with their favourite books. Listing 6.7 presents on the left-hand side an example query document to get some basic information about the person named *Alice*. The requested data includes her name and age, which is renamed to the property `years` in the result set. The query includes her favourite books with their title and authors. The string-based format of the request message presented in the example strictly follows the GraphQL specification, though whitespace and commas are syntactically and semantically insignificant. The right-hand side of Listing 6.7 presents an example result for this query as a JSON document. Its structure and entities have to follow the request: only keys which were specified in the query are allowed to be part of the result document.

6.4.2. The GraphQL Type System

Being part of the data access and manipulation layer, GraphQL does not support ad-hoc queries like most standard database drivers provide. Instead, queries have

Listing 6.8: Type definitions for the example query of Listing 6.7 and GraphQL's built-in `Query` type.

```
type Query {
  person(name: String!): Person,
  book(title: String!):  Book,
  books(filter: String): [Book]
}

type Book {
  title:   String!,
  authors: [Person]
}
```

```
type Person {                                    GraphQL
  name:    String!,
  age:     Integer,
  friends: [Person],
  books(favourite: Boolean): [Book]
}
```

to satisfy the type system previously defined by the server administrator. The SDL is expressive and supports features like inheritance, composition, interfaces, lists, custom types, and enumerated types. By default, every type is nullable, i. e., not every value specified in the type system or query has to be provided, unless denoted by the exclamation mark `!`.

Every GraphQL type system must specify a special root type called `Query`, which serves as the entry point for the query's validation and execution. In Listing 6.8, we present a minimal definition of a type system to satisfy the example query of Listing 6.7. It defines the two types `Person` and `Book` as objects. Their fields can have arguments. For instance, when retrieving the books of a particular person, it can be specified to return only their favourite books by providing an appropriate Boolean flag. A detailed introduction to GraphQL's type system is out of scope of the section at hand. Nevertheless, our GraphQL framework for SWI-Prolog in [84] also implements more complex properties of the type system.

6.4.3. Integration with Quasi-Quotations, DCGs, and Dicts

The working draft of GraphQL specifies the syntax for query documents as well as for the type system in the form of a formal grammar. For integrating GraphQL into SWI-Prolog as an external DSL, our implementation makes great use of quasi-quotations, definite clause grammars, and dicts (cf. Section 3.3.5). Quasi-quotations are used to directly embed the query document as well as the type definitions in the Prolog source code. In order to define single types and the schema, our implementation provides several quasi-quotation tags, for example `schema/4` for the overall schema, and `type/4` for the declaration of a single type, which use plain old DCGs to parse the contained strings. After all, the grammar specified in the GraphQL specification

Listing 6.9: Definition of the `type` quasi-quotation.

SWI-Prolog

```
1  :- use_module(library(quasi_quotations)).
2  :- quasi_quotation_syntax(type).
3  type(ContentHandle, _SyntaxArgs, Vars, Res) :-
4    phrase_from_quasi_quotation(gql_type(Vars, Res), ContentHandle).
5
6  gql_type(Env, Type) -->
7    #, type_definitions(Env, Definitions), #,
8    { Type = object{}.put(description, _)
9                     .put(fields, Definitions) }.
10 type_definitions(Env, Definitions) --> % base case
11   type_definition(Env, Name, Type),
12   { Definitions = _{}.put(Name, Type) }.
```

for query and type documents has been translated into a DCG, resulting in more than 50 nonterminals just to parse queries. The entire DCG rule base to parse the type system as well is implemented in more than 600 lines of Prolog code with more than 100 nonterminals.

An extract from the implementation of the quasi-quotation for `type/4` is given in Listing 6.9. It uses the DCG nonterminal `gql_type//2` to parse the given type, and binds the variable `Res` to an internal Prolog representation based on dicts. Its `description` property (l. 8) is optional and used only for debugging purposes in GraphQL. The nonterminal `#//0` consumes the semantically insignificant whitespace and commas; `type_definition//3` parses a single key-value expression and is left here for the sake of simplicity.

After parsing the quasi-quotations using DCGs, the GraphQL schema and types are represented by dicts. In Listing 6.10, we present an extract of the generated dict for the type `Person`. Since SWI-Prolog's compiler calls for quasi-quotations provides access to variables of the quasi-quotation's outside-world context, all embedded user-defined type names starting with an uppercase letter create corresponding Prolog variables, therefore allowing cross-referencing of types within a GraphQL schema. The type `Person`, which is recursive in the field `friends`, can therefore be created as stated in line 1, similarly resulting in an infinite and cyclic Prolog term. Since only GraphQL's types are allowed to be cyclic but not the queries, this will not lead to non-termination on execution.

In order to execute a query, both the dict generated for the query and for the type system are traversed simultaneously in a top-down approach. Beginning with the root type `Query` specified in the GraphQL schema, our system searches for the requested fields in the type definitions. To get the appropriate value, for every type

Listing 6.10: Generated dict for the GraphQL type `Person`.

```
?- Person = {|type|| friends: [Person], ... |}.
Person = object{
  fields: _{
    name: field{ type: string, nonNull: true, resolve: _ },
    age: field{ type: integer, resolve: _ },
    friends: list{ kind: field{ type: Person }, resolve: _ } },
    books: list{
             arguments: _{ favourite: field{ type: boolean } },
             kind: field{ type: Book }, resolve: _ },
  },
  resolve: _ % not yet bound
}
```

Toplevel SWI-Prolog

of the schema a `resolve/5` predicate has to be defined, which was initially a free variable in Listing 6.10. This *resolver* is used to generate the resulting dict for a specific type, which is the basis for the JSON document returned to the client. For a detailed description of the resolving mechanism, we refer to our work [84, Sec. 5]. It allows to handle data of various sources and to combine, e. g., relational with NoSQL databases, resulting in GraphQL as a single unified query language for deductive databases.

The definition of the GraphQL type system in the same way it is used in the GraphQL working draft, i. e., as a domain-specific language in Prolog, enables developers who are not yet familiar with Prolog to build a custom GraphQL server. In addition, it is possible to specify an application layer just by the DSL, separated from the actual data access that is implemented by Prolog resolvers. Even for developers who are experienced with Prolog, the notation in the form of a DSL results in a short and readable data and query description, which might also serve as a source format for query validation and optimisation.

7

A Tracing Meta-Interpreter for Web-based DCG Visualisation

I hate almost all software. It's unnecessary and complicated at almost every layer. The only software that I like is one that I can easily understand and solves my problems. The amount of complexity I'm willing to tolerate is proportional to the size of the problem being solved. The only thing that matters in software is the experience of the user.

— Ryan Dahl[30]

Compared to the definition as an internal DSL which just relies on some operators with appropriate types and precedences, the required Prolog source code to integrate the same language as an external DSL – in most cases using a definite clause grammar – easily becomes much larger. Even without the modelling of additional and grammatical constraints right into the grammar, the DCG for our simple DSL for the specification of if-then rules alone constitutes a complex software system.

However, regardless of its long history and wide distribution, there are rare tools dedicated to the work and development of DCGs. Here, its close relation to Prolog has its drawbacks: it is often recommended to use Prolog tracers and debuggers, although their representation and available functions do not suit well for the special-purpose DCGs. Having to work with tools created for Prolog creates unnecessary overhead and requires deep knowledge about this programming language, which is actually not necessary to just understand a given DCG or its application.

In this chapter, we present an interactive, web-based visualisation and tracer for DCGs. It can be useful both for beginners who learn DCGs, as well as for experienced users debugging their grammar. As part of our contribution, we discuss different techniques to collect information about the execution of a DCG in SWI-Prolog, including term expansions, trace interceptors, and meta-interpreters. After

[30]Shortened quote from "I hate almost all software" (2011), Google Plus blog post originally located at https://plus.google.com/115094562986465477143/posts/Di6RwCNKCrf, archived version available at https://gist.github.com/cookrn/4015437. Ryan Dahl is an American software engineer and the original developer of the Node.js JavaScript platform.

all, our system is a feature-rich, self-contained example application which illustrates how to develop Prolog applications for the web using SWI-Prolog's *Pengines* [70] infrastructure. The system was created with Jona Kalkus as part of his master's thesis [59] at University of Würzburg, Germany. The source code of our tool is published under MIT License at https://github.com/fnogatz/dcg-visualiser. It requires SWI-Prolog of at least version 7 (2014). Although its concepts can be tailored for other and older Prolog systems as well, our implementation relies on some of the extensions recently added to SWI-Prolog, resulting in a more compact Prolog program. For instance, we use dicts for JSON creation, and SWI-Prolog's *Pengines* library to create a web-based Prolog application.

The chapter at hand is based on our corresponding publication [81]. Technical aspects of the client-server architecture and the inter-process communication are described in more detail in this publication, as well as a literature review of related work regarding interactive Prolog debuggers and program visualisations. In this chapter on the other hand, we focus on the adaptions of the previously introduced techniques to use and transform DCGs in order to collect all information about their execution at run-time that are needed for a useful visualisation and tracing. The system's aim is to assist with the development of large definite clause grammars, as they are common and necessary when defining expressive Prolog DSLs externally. It has proven useful in the development of our *library(plammar)*, a large Prolog grammar written in Prolog, which is presented in detail in Chapters 9 and 10.

The remainder of the chapter at hand is organised as follows. In Section 7.1, we define the demands on an interactive tracer dedicated to DCGs. The resulting system and its web-based interface is presented in Section 7.2 using the example of tracing the parsing of if-then rules. Next, Section 7.3 discusses several approaches to collect the information needed for this visualisation from the remote Prolog process. The chapter concludes with an overview of the system's overall client-server architecture using *Pengines* in Section 7.4.

7.1. Important Criteria for an Interactive Visualisation

Since the early years of Prolog, there has been research on debuggers and program visualisations. In the extensive survey on logic program analysis and debugging of Ducassé and Noyé from 1994 [33], an overview of existing approaches at that time is given. These and more modern tools and approaches are usually focussed on the work with Prolog in general, so at least they already handle common techniques like backtracking and unification.

However, none of the existing tools can be easily used to visualise the processing of character-based difference lists with DCGs in an intuitive way. We therefore create a web-based tool to accomplish the following needs. To the best of our knowledge, there is currently no tool that already fulfils these demands.

Focus on DCGs. The tool for execution and visualisation is focussed only on the application of definite clause grammar rules. Since DCGs are often translated into plain old Prolog clauses, only the expanded forms of the grammar rules are available in existing graphical tracers and visualisations. Therefore, the shown line numbers do not refer to the grammar rule's original location, and the tracer's output is more verbose, as it contains both additional, implicit arguments of the DCG that represent the difference list. In addition, the expanded form makes no difference between Prolog predicates that originate from expanded nonterminals and Prolog code that was enclosed in curly brackets – existing visualisations simply mix both grammar rules and embedded Prolog predicates.

Compatibility and Feature-completeness. Existing DCGs are supported out of the box, without the need for further modifications to be compatible with the tool for visualisation and tracing. Several existing solutions, for instance the *Prolog Visualizer*,[31] define a meta-interpreter for Prolog or grammars from scratch. Although these interpreters are often easier to adapt for tracing, the users are restricted to the limited subset of the original language as defined in the meta-interpreter. For instance, contained user-defined operators cannot be used as they are not provided by the original meta-interpreter.

DCGs are most likely applied with difference lists of characters. They should be presented in a user-friendly format as strings. Nevertheless, grammars that describe lists of any other Prolog data structures should also be supported.

Reasonable Performance. Collecting data for tracing always results in an overhead. For instance, static code analysis and optimisations which are normally performed at compile-time should be avoided, as they change the executed source code. Nevertheless, it is intended to reach a reasonable performance even with activated tracing.

Interactive Exploration of Rule Applications. Some existing tools generate just static images or dependency graphs based on static code analysis. Our system can be used interactively in a web-based environment.

[31] The *Prolog Visualizer* by Lai and Warth (2015) is a web application which features an in-depth step-by-step execution of Prolog. A public instance is available at https://cdglabs.org/prolog/.

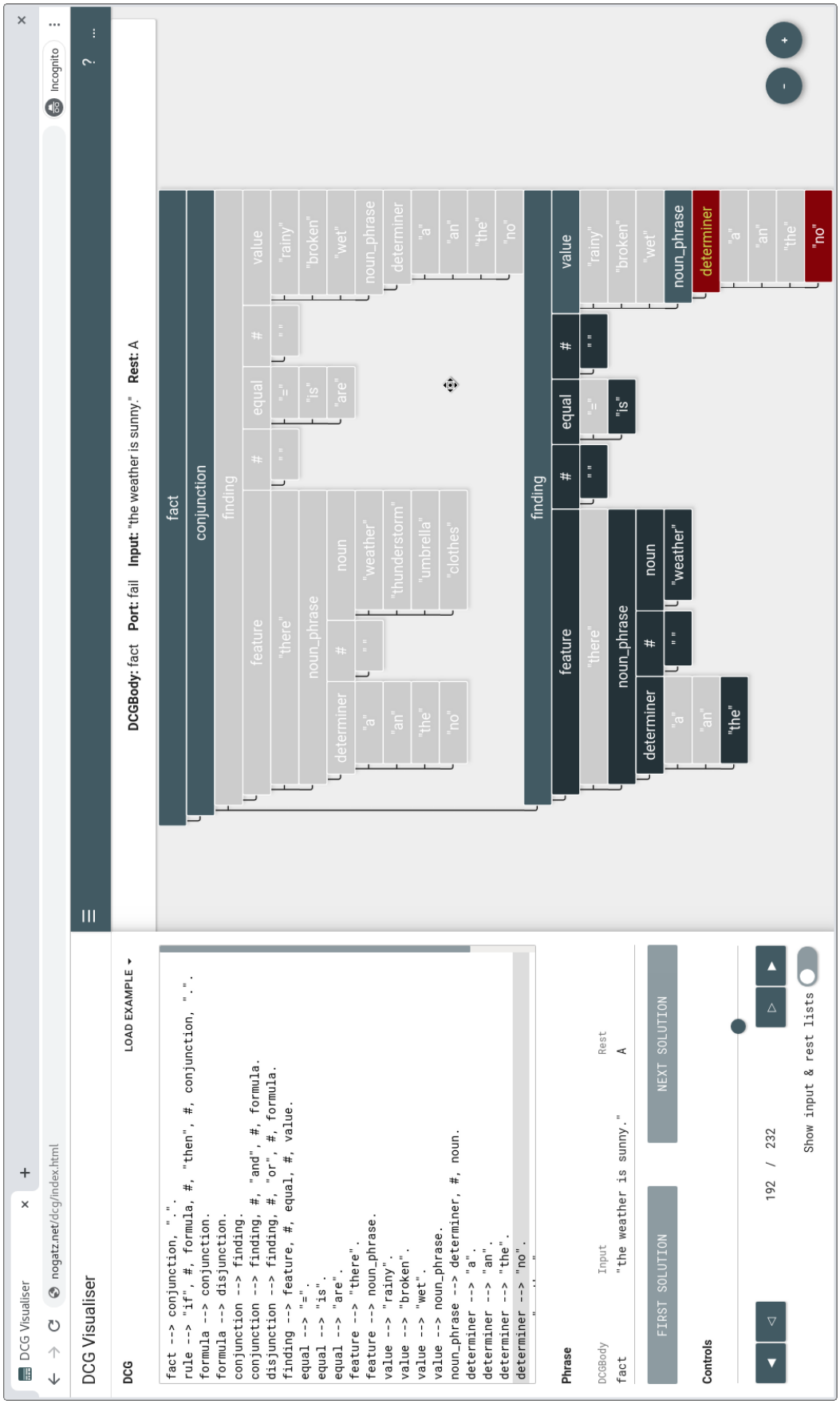

Figure 7.1: Screenshot of the web-based interface for DCG visualisation and tracing.

This list makes no claims for completeness. However, it should cover all essential aspects that assist with the development of grammars using DCGs.

7.2. User Interface and Example Application

Figure 7.1 presents a screenshot of the created web application opened in the *Google Chrome* web browser. The interface and its functionality is inspired by collaborative code sharing platforms like SWI-Prolog's SWISH [7]. On the left-hand side, the code editor is used to edit DCGs. We provide several examples to present the different features of this tracer, which can be loaded via the *Load Example* dropdown element. In the *Phrase* section in the left part of the application, the user can specify the parameters for the `phrase/3`$_{\texttt{ISO}}$ predicate. The execution can be replayed step-by-step and using a slider in the *Controls* section.

In the right panel, the execution is interactively visualised. Only nonterminals and terminals are displayed here. By activating the setting *Show input & rest lists* in the *Controls* section, the processed difference lists are additionally displayed in the nodes of the search tree. The difference lists and nonterminals are written as strings in case they represent lists of characters. Otherwise, the list elements are printed as arbitrary Prolog terms.

By hovering over the elements on the right-hand side, the corresponding source code fragment (which might contain arbitrary Prolog code) gets highlighted in the *DCG* textbox in the application's left part. Backtracked alternatives are highlighted in grey, failing rule applications are marked in red.

In the example screenshot of Figure 7.1, we used the DCG of Section 6.3 to describe if-then rules as an external Prolog DSL. Instead of logical disjunctions via `;/2`$_{\texttt{ISO}}$, alternatives are written as multiple grammar rules with the identical nonterminal in the rule's left-hand side. This allows for a better visualisation and source code highlighting when tracing the backtracking of alternatives, as otherwise only parts of a grammar rule's right-hand side should be highlighted, which would require the client-side parsing and analysis of the given DCGs. From a semantic point of view, this notation of disjunctions is equal to the DCG to parse and serialise if-then rules we presented in Listing 6.5. Nevertheless, our tracer supports both notations for alternatives.

The screenshot illustrates step 192 of 232 in the execution of the overall failing goal `?- phrase(fact, "the weather is sunny.", A)`. A `fact//0` is known to be a `conjunction//0` followed by the full stop `.` as a terminal symbol. In the upper part of the search tree, the first alternative for `conjunction//0`, the grammar rule

`conjunction --> finding`, is applied. It does not succeed, because `sunny` cannot be parsed by the nonterminal `value//0`: it is none of the terminals `rainy`, `broken`, `wet`, and on the other hand no valid `noun_phrase//0`, as it does not start with one of the valid determiners `a`, `an`, `the`, and `no`. As a result, all previous steps are backtracked and thus marked by a grey background, and the second alternative for `conjunction//0` is tested in the lower part of the shown search tree. In the currently depicted step 192 of 232 of the execution, the last alternative for the nonterminal `determiner//0` is tested. Since it fails for the given input `sunny`, the node for the nonterminal `determiner//0` is marked by a red background. In the next steps, all other alternatives are applied by backtracking, thus consecutively highlighting the failing nodes. After all, the initial goal `?- phrase(fact, "the weather is sunny.", A)` is known to be failing.

Note that in the DCG we presented in Listing 6.5 the two alternative grammar rules for the nonterminal `conjunction//0` have the same prefix. As a result, it is actually known that none of them succeeds as soon as the nonterminal `finding//0` fails. This can be made explicit by modifying the grammar rule for `conjunction//0` and introducing a new one that first parses the common prefix, followed by a newly created nonterminal `conjunction_f//0` which is either empty or the rest of a conjunction of the form `· and ·`:

```
1 conjunction   --> finding, conjunction_f.                    PROLOG
2 conjunction_f --> [].
3 conjunction_f --> #, "and", #, formula.
```

This technique is called *factoring out*, and a common transformation of grammars to resolve conflicts in *LL(1) parsers*, or to optimise *LL(k)* and *LR(k) parsers* (cf. Section 5.6). We do not elaborate on this and other optimisations for grammars, as our tool just visualises the execution of a DCG as it is.

This visualisation can grow fast for long sentences or big DCGs. Therefore the panel on the right-hand side of our application supports zoom and scrolling, similar to web-based map applications.

7.3. Collection of Run-Time Information

In order to implement the targeted visualisation, some information related to the grammar rules and its execution have to be collected. Due to recursive rules, the complete language space of a DCG can be easily infinite. To avoid handling and visualising infinite trees, our approach is not only based on the DCG itself, but on

the execution of a `phrase/3` query using this grammar. The main goal is to produce a parse tree for a given query. However, two difficulties appear when we produce a near to complete visualisation: (i) the exact execution order has to be stored, and (ii) including also the failing branches and the performed backtracking. The latter is rather complicated, since in Prolog we usually only have information about succeeding rule applications rather than failing ones. In the following, we present two approaches to extend a given DCG to collect the required data and discuss their advantages and disadvantages.

7.3.1. Intercepting the Built-in Tracer

SWI-Prolog provides a command line based tracer which can be enabled by calling the built-in predicate `trace/0`$_{\texttt{SWI}}$. Following Prolog's 4-port execution model [14], every *call*, *exit*, *redo*, and *fail* step gets printed. In addition, the user is asked whether to proceed, or skip the tracing of some subgoals, or to stop. SWI-Prolog extends the 4-port model by two additional ports, *unify* and *exception* [140, Sec. 3.6], resulting in a 6-port model with more events that can be traced.

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 and is not well suited for further program based analysis. Therefore SWI-Prolog offers the possibility to redefine the tracer itself, thus allowing a more explicit and detailed control and handling of the traced events. The predicate `prolog_trace_interception/4`$_{\texttt{SWI}}$ is a built-in hook which is called from the SWI-Prolog debugger before writing the trace statements to the command line. Provided that the call of this user-defined predicate succeeds, the built-in text-based tracer is intercepted [136, Sec. B.3]. The hook provides information about the current port, stack frame, and choice points. From the stack frame, additional information can be retrieved, for instance about the source code location of the currently used clause. By defining the hook via the predicate `prolog_trace_interception/4`$_{\texttt{SWI}}$, we can store detailed information about the actual execution of the given DCG.

In its simplest form, the data required for the DCG visualisation can be asserted with the dynamic predicate `step/`n. An excerpt of such a custom trace interceptor is given in Listing 7.1. For each step in the tracer, a fact `step/6` is asserted. It contains a unique, consecutive number, and the port according to SWI-Prolog's 6-port execution model. As an example, we additionally store information about the current event: the frame number, the currently executed goal, and its parent. `Choice` is a reference to the last choice point. With `prolog_choice_attribute/3`$_{\texttt{SWI}}$, properties can be examined, like its frame number. `Action` has to be unified with a

Listing 7.1: Definition of a custom trace interceptor to collect execution information in the dynamic predicate `step/`n.

```
prolog_trace_interception(Port, Frame, Choice, Action) :-
  get_counter(N), % retrieve unique program counter
  % get information about current event
  prolog_frame_attribute(Frame, goal, Goal),
  prolog_frame_attribute(Frame, parent, Parent),
  prolog_choice_attribute(Choice, frame, ChoiceFrame), % ...
  % assert information as a dynamic predicate 'step'
  assert(step(N, Port, Frame, Goal, Parent, ChoiceFrame)),
  % continue with the execution
  Action = continue.
```

PROLOG

term that specifies how to continue with the execution. It is typically retrieved by asking the user. Typical values are `abort`, `continue`, and `nodebug`.

Each time the tracer is invoked, a fact `step(N,...)` is asserted. With the help of its unique, incrementing program counter `N`, the program execution can be reproduced in detail. The usage of this tracer in SWI-Prolog is identical to starting the normal, built-in tracer. The asserted facts of the predicate `step/`n can be retrieved and displayed like any other, e. g., by using `listing/`$\{1,2,3\}_{\texttt{SWI}}$ (cf. Section 3.5):

```
?- trace, phrase(noun_phrase, NP), notrace, listing(step).
NP = "a weather" .
% step(1,...) and step(2,...)
step(3,unify,235,noun_phrase(A,[]),call_dcg(noun_phrase,A,[]),...).
step(4,call,255,determiner(A,B),noun_phrase(A,[])),...).
% ... following steps that lead to the binding of A and NP
```

TOPLEVEL

This example shows two asserted facts. Firstly, the unification of the goal `?- noun_phrase(A, [])` with a matching rule head, followed by the execution steps required for satisfying `determiner//0`, which is the first item in the grammar rule body of the nonterminal `noun_phrase//0`.

One of the advantages of using a custom trace interceptor 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 reimplementing any logic, or even a complete meta-interpreter. In addition, this offers the advantage that built-in predicates can be evaluated without additional efforts. However, we encountered a major drawback in this approach: SWI-Prolog's frame number is not unique, so there are cases in which the same frame reference is used by different goals. This has been observed in the context of deterministic goals. Execution frames which

cannot generate further solutions are removed from the execution stack and their frame reference is thus released. Therefore, this approach is not reliable enough for the visualisation of DCG execution.

7.3.2. Automatic Generation of Parse Trees

It is possible to extend a given DCG to hold information about the currently used rule by adding an additional argument to every nonterminal. This way, a parse tree is generated on-the-fly while parsing the given input. A modification in this way does not alter the core semantics of the given grammar rules. Only calls of `phrase/`$\{2,3\}_{\texttt{ISO}}$ have to be slightly adapted to hold the additional argument for the parse tree. For instance, `?- phrase(fact(Tree), "the weather is rainy.")` is called instead of using the nonterminal `fact//0`.

We discuss this modification of DCGs in detail in Chapter 8 and present a transformation of grammar rules at compile-time using term expansions. The generated parse tree can then be displayed graphically, depicting the order of the execution in the tree structure, from top to bottom and left to right, similar to depth-first search. However, this approach supports only succeeding rule applications: if a grammar rule cannot be applied or is backtracked, this is not depicted in the parse tree, following Prolog's SLD resolution mechanism. Therefore, the extension of DCGs as introduced in Chapter 8 does not fit our requirement to also illustrate failing branches of the grammar's execution.

7.3.3. Modified Meta-Interpreter

Another mean to trace the execution of a DCG is by implementing an appropriate meta-interpreter that stores the required information. In Section 6.2.3, we presented an implementation of `phrase/3`$_{\texttt{ISO}}$ which does not rely on the term expansion of the grammar rules, but instead describes their application and the handling of all contained control structures that are allowed in the rule's bodies on the right-hand side. The complete meta-interpreter is given in Appendix B.4.

The meta-interpreter again implicitly handles Prolog's backtracking, thus it cannot be traced easily. One approach to avoid this limitation is to reimplement the logic of backtracking in the meta-interpreter, too. This has been done by Dave Bowen for generating AND/OR trees.[32] However, it requires extensive changes to the original

[32] http://www.dcs.ed.ac.uk/home/mke/edinburgh/tools/tracing/andor.pl

meta-interpreter of `phrase/3`$_{\texttt{ISO}}$. Not only have already collected data and pending goals to be passed within the meta-interpreter, also basic mechanisms such as unification can no longer be used.

Another approach is to store information during execution in a way that is not affected by backtracking. For instance, it is possible to simply print information about the current step to the user. Alternatively, the built-in predicate `assert/1`$_{\texttt{ISO}}$ can be used again. In this way, the meta-interpreter can use backtracking and unification mechanisms, and still obtain information about failed execution steps. They are then stored in a dynamic predicate, similar to the assertion of `step/`n predicates as discussed for the trace interceptor in Section 7.3.1.

To have a fallback on all applications on the right-hand side, the goals on the right-hand side are wrapped in a disjunction, i. e., `Goal` becomes `Goal ; print_indented(Goal:fail, L)`, where the predicate `print_indented/2` is used to log a failing grammar body at the level `L`. In our meta-interpreter, this wrapper is used for every grammar rule's right-hand side which might be backtracked or never succeed, so it will be nevertheless shown in the tracer. For instance, the original goal `?- call(P)` for a Prolog goal `P` embedded in a grammar rule (cf. line 6 of Listing 6.2) is expressed as follows:

```
1 phrase({ P }, L, A, A) :-
2     call(P), print_indented(call(P):exit, L)
3   ; print_indented(call(P):fail, L).
```

For every successfully computed answer substitution for the goal `?- P`, the predicate `print_indented/2` is called with the term `call(P):exit` as its first argument. Only if there is no (further) solution, it is called with the argument `call(P):fail`, indicating the failing query. The complete modified meta-interpreter for DCGs with tracing is given in Appendix B.6.

We extend the predicate `phrase/3`$_{\texttt{ISO}}$ by an additional argument `L` which stores the current level of execution. It allows to limit possibly infinite recursions, as well as to print nicely formatted and indented trace outputs. As a basic example, we use `print_indented/2` to print the tracer events to the user:

```
print_indented(A, L) :- I is 2*L, tab(I), writeln(A).
```
PROLOG

Here, SWI-Prolog's built-in predicate `tab/1`$_{\texttt{SWI}}$ writes the given amount of spaces to the current output stream.

When querying a DCG, the resulting structure is displayed as a sideways tree, with child elements being more indented. The single lines are of the format

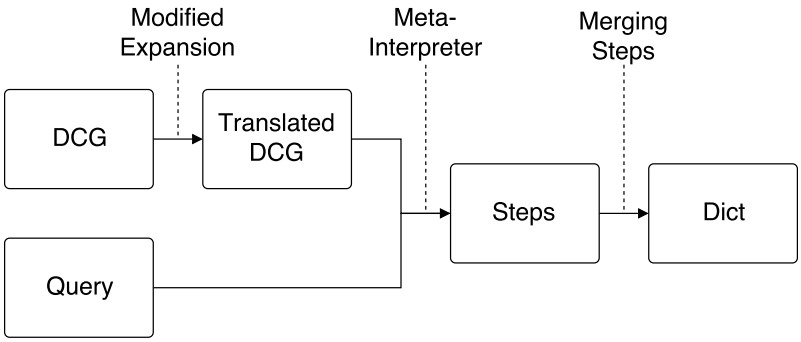

Figure 7.2: Server-side components to generate the trace data.

`<goal>:<call|exit|fail>` or in the case of unifications `<unification>:<exit|fail>`. Cuts are indicated by the atom `cut`, and the processing of terminal symbols is denoted by `empty` and `T:consume`, respectively. Failing clauses are also printed before calling available alternatives. Thus, an output is also possible if a subgoal has no successful answer substitution.

7.4. Client-Server Architecture with Pengines

The definition of the predicate `print_indented/2` can be easily changed to assert `step/`n facts again. They are enriched by information about, among others, the original source code location of the called rule. At the end, all the asserted facts are collected and sent back to the web-based tracer with the help of SWI-Prolog's *Pengines* framework, where they are visualised and shown to the user.

Modified Term Expansion. By default, SWI-Prolog translates DCGs into plain old Prolog clauses via term expansion (cf. Section 6.2.4). Since our meta-interpreter for `phrase/`$\{2,3,4\}_{\texttt{SWI}}$ relies on their original representation, we bypass SWI-Prolog's `dcg_translate_rule/2`$_{\texttt{SWI}}$ by defining a user-defined term expansion for `-->/2` which is executed first. It wraps the DCGs into a new predicate instead, e. g., `--->/2`.

Dict Generation. The server-side components that generate the data required for the visualisation are shown in Figure 7.2. A query is evaluated by the modified meta-interpreter as presented in Section 7.3.3, with information about the individual execution steps being temporarily stored as Prolog facts. Then, these facts are merged into a single Prolog term, that is used to send the tracing information to the client. We use dicts, because they allow for easy serialisation into JSON, as introduced in Section 3.3.5.

Server Architecture. *Pengines* [70] offers the means to use logic programming in web-based projects. It is based on SWI-Prolog's libraries for threads, HTTP clients and servers. *Pengines* implements a universally applicable high-level interface between SWI-Prolog and clients. The underlying observations of the conversations taking place between Prolog and a user resulted in the definition of the *Prolog transport protocol* and its communication protocol over HTTP ($\text{PLTP}_{\text{HTTP}}$). For a more detailed introduction to PLTP, we refer to Section 4.3.6.

Our DCG visualisation is based on the demo server of the *Pengines* project. On the server-side, we use the previously presented modified term expansion and DCG meta-interpreter. For every user, a new *Pengines* thread is created with a separate `user` module. This way, the facts are asserted in a local scope, and are destroyed when the user disconnects. It prevents the accumulation of unnecessary facts inside the meta-interpreter's module, enables concurrent requests by multiple *Pengines* threads, and ensures security.

To prevent the execution of possibly malicious Prolog code, the Prolog code embedded in the DCGs is checked before its execution, using *Pengines*' built-in security features. Allowed Prolog predicates can be declared using the predicate `sandbox:safe_goal/1`$_{\texttt{SWI}}$.

Client Architecture. The web application is based on common web technologies: JavaScript, HTML5 and CSS3. In addition, several frameworks are used in our implementation. The layout of the application is based on *MUI*, which is a lightweight CSS framework based on Google's Material Design Guidelines. Furthermore, *CodeMirror* is used as an online text editor for DCGs.

Once the user has entered a DCG and formulated a request, it is executed on the server-side with the help of *Pengines* on SWI-Prolog. The dict generated by the meta-interpreter is returned as JSON. It is rendered using the logicless template engine *Handlebars*. Interactive features, such as displaying individual execution steps, are realised with the help of *jQuery*.

8

Automatic Parse Tree Generation for Definite Clause Grammars

We should forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3 %. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

— Donald Knuth[33]

Definite clause grammars provide a powerful and expressive formalism to describe external Prolog DSLs by a context-sensitive grammar. As a running example throughout our work, we defined the declarative if-then rules as an external DSL in Chapter 6, and discussed its interactive tracing and visualisation when parsing a given input sentence in Chapter 7. However, the practical benefits of this DCG are limited: without further modifications, it can only be used to check *whether* a given string can be parsed by the grammar, but provides no additional information on *how*. And though the same DCG can often be used to generate sentences that conform to the grammar – because Prolog makes no difference in input and output arguments, and `phrase/3`$_{\text{ISO}}$ also allows calls to the predicate in the mode `(:,-,-)` –, the format and order of the difference lists returned via backtracking cannot be controlled. Since the application of `phrase/3`$_{\text{ISO}}$ and the provided grammar rules strictly follow Prolog's SLD resolution mechanism, the computed answer substitutions depend only on the grammar rules' and terminals' order of appearance in the source code. In particular for left-recursive grammar rules, this easily results in correct yet not meaningful sequences of valid sentences, when backtracked over free variables. For instance, the EBNF grammar that describes a valid variable token in Prolog as presented in Figure 5.2 would create the possible variable names in the order of `_`, `_a`, `_aa`, `_aaa`,

[33] Donald Knuth is an American computer scientist, mathematician, and professor emeritus at Stanford University. He created several programming languages, including the typesetting system TEX. In 1974, Donald Knuth was awarded the Turing Award for his monograph "The Art of Computer Programming", which since then became a reference work in the field of algorithms and their analysis, and the quote is taken from.

and so on, instead of the possibly more intended and descriptive sequence starting with `_`, `_a`, `_b`, and `_c`.

Therefore, for practical applications, the execution of the DCG should result in a corresponding Prolog term that represents the parsed sentence. In the second case of serialisation, i. e., mode `(:,-,-)` of `phrase/3`$_{\text{ISO}}$, this Prolog term can be consumed to create the corresponding difference list. In its simplest form, the Prolog term is a parse tree that contains the name of the used nonterminal and its arguments, i. e., the grammar rule's left-hand side. This extension of a DCG has been proposed for natural language processing in the past and can be done automatically as a Prolog program transformation using term expansions.

In this chapter, we first describe in Section 8.1 possible representations of the parse tree as nested Prolog data structures, and the classical approach to use compound terms. In the following Section 8.2, we discuss the general methods to pass this parse tree term on to and from the grammar rule's execution context. The presented modification of an existing DCG to additionally process the corresponding parse tree is generic, and can be applied as a source-to-source transformation at compile-time. We present and discuss this preprocessing term expansion scheme for DCGs with parse trees in Section 8.3. It also introduces the SWI-Prolog package *library(dcg4pt)*, where all of this functionality has been wrapped up. The *library(dcg4pt)* acts as a drop-in replacement for Prolog's traditional term expansion scheme for DCGs, which we introduced in Section 6.2.4, so that our contribution additionally implicitly handles the processed parse tree. Section 8.4 extends these considerations to optionals and sequences of nonterminals, which likely appear in descriptions of formal languages, and are therefore of particular interest when parsing and serialising external DSLs in Prolog. The variadic number of processed terminals usually introduces possible non-termination, which needs to be handled to preserve the grammar's ability to be used both for parsing and serialisation, i. e., to create a parse tree by a given source code and vice versa. The chapter concludes in Section 8.5 with an overview of alternative approaches to the generation of parsing trees, and other extensions for DCGs and formal grammars in Prolog.

The chapter at hand extends our work of the corresponding publication [85, Sec. 3–5], where the foundations of *library(dcg4pt)* have been introduced in a shorter form:

> Falco Nogatz, Dietmar Seipel, and Salvador Abreu. Definite Clause Grammars with Parse Trees: Extension for Prolog. In *8th Symposium on Languages, Applications, Technologies (SLATE 2019)*, volume 74 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:14, 2019.

The created SWI-Prolog package *library(dcg4pt)* is published under MIT License at https://github.com/fnogatz/dcg4pt. It is available for installation from SWI-Prolog's list of add-ons.

8.1. Representing an External DSL as a Prolog Term

The general idea of parse trees is to extend a given DCG so that for every application of a grammar rule some information about the corresponding nonterminal and its argument is stored in a structured Prolog term. This way, a parse tree is generated on-the-fly while parsing the given input. The Prolog representation can be of any type that allows nested terms, e. g., dicts, or association lists like the *field notation* of Seipel et al. [104, 105]. We use compound terms, as they can be constructed and processed just by predicates provided by the ISO Prolog standard, without the need for a particular Prolog system or additional libraries.

The nested, compound Prolog term that represents the parse tree can be displayed graphically. Then, the order of the execution – given it was created by normal SLD resolution with `phrase/3`$_{\texttt{ISO}}$, and not by using a meta-interpreter – as well as the applied grammar rules can be read from the tree structure. The term represents the program flow from top to bottom and from left to right, similar to a depth-first search. Unlike a binary expression tree (e. g., as depicted for an exemplary if-then rule in Figure 5.1), terminals appear only in the parse tree's leaves, while the nodes depict only nonterminals.

The parse tree can be easily extended by further information, e. g., about the source code locations of the applied grammar rules. However, only succeeding rule applications are part of the created Prolog term representation – if a grammar rule cannot be applied or is backtracked, this is not depicted in the tree, following Prolog's SLD resolution mechanism that returns only for succeeding paths in the search tree. Since the parse tree is just an internal Prolog representation of the external DSL, it is not of interest to additionally store information about failing applications of grammar rules. For use cases where these are required, e. g., for debugging purposes, we refer to our modified tracing meta-interpreter for DCGs of Chapter 7.

In Figure 8.1, we give a graphical visualisation of the parse tree that represents the external DSL's fact `the weather is rainy.` when parsed using the DCG for if-then rules according to Listing 6.5. The inner nodes depict the resolved nonterminals, while the terminals are given in the parse tree's leaves as strings and highlighted in grey. For the sake of simplicity, we omit the handling of whitespaces, i. e., the presented parse tree misses all occurrences of the nonterminal `#//0`.

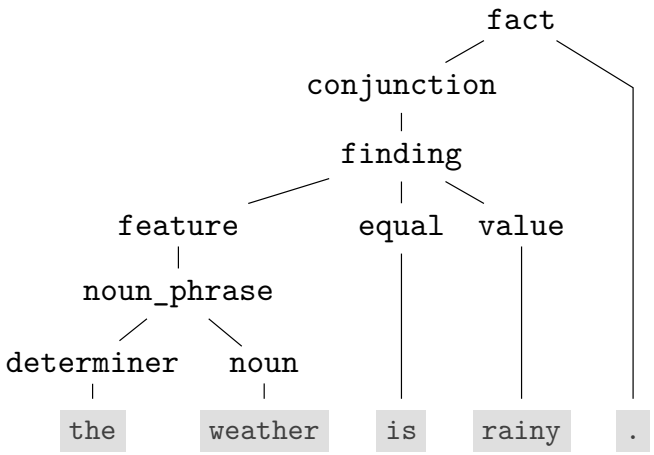

Figure 8.1: Parse tree of the sentence `the weather is rainy.` for the nonterminal `fact//0` using the DCG that describe if-then rules.

Following the ideas proposed by Abramson and Dahl [2], the parse tree argument for a DCG rule `H --> B` can be represented by a compound term of the form `H(T)`, with `H` being the name of the rule's head without arguments, and `T` being a term whose structure depends on the rule's body `B`. Given that `B` is a sequence of n terminals and nonterminals, we can represent each as an argument to the term `H`, thus the parse tree for the grammar rule `H --> B1, ..., Bn` is represented by a parse tree of functor `H/`n. Original arguments of the grammar rule's left-hand side `H` are not depicted in the corresponding parse tree. This list of arguments could be added as the parse tree's first argument. In the following as well as in our *library(dcg4pt)*, we refrain from this additional distinction of the applied grammar rule to keep the processed parse tree concise. Thus, it only depicts the name of the resolved nonterminal together with the elements of the grammar rule's right-hand side.

8.2. Process Parse Trees in DCGs with State Passing

In comparison to EBNF, DCGs provide three extensions that allow to describe context-sensitive grammars, as previously summarised in Section 5.6: (i) complex control structures in the grammar rule's right-hand side, (ii) arguments in the grammar rule's left-hand side, and (iii) the semicontext notation with pushback lists to prepend elements to the list of processed terminals. All of them can be used to create or consume a parse tree in the process of parsing or serialisation. However, control structures require global variables using dynamic predicates as introduced in Section 3.7 to store and modify the current state of the parse tree when executing the DCG. Global variables are therefore hardly suitable to pass the parse tree from one grammar rule to another. In this section, we compare the other two means, namely

arguments and pushback lists, with respect to their use with parse trees, and discuss their properties and call semantics.

8.2.1. Comparison of Context-Sensitive DCG Extensions

In contrast to the use of global variables, both context-sensitive DCG extensions of additional arguments in the nonterminals as well as pushback lists allow to encode the parse tree right into the DCG notation. Then, this compound Prolog term can be passed around and modified as a state representation when applying the grammar rules:

Additional Argument. In the context of natural language processing, it has been proposed to extend the grammar rule's left-hand side by an additional argument that holds the parse tree. This way, the nonterminal `finding//0` becomes `finding//1`. For nonterminals with an arity greater than zero, the convention is to add the parse tree as the new very last argument.

Semicontext Notation. With the semicontext notation of DCGs, it is possible to push back elements to the processed list. Similar to a nonterminal's additional argument, this allows to pass a Prolog term from one applied grammar rule to another. Instead of relying on the previously mentioned convention that the parse tree is given in the nonterminal's last argument, it is instead expected to be the nonterminal's immediately following list element. As a consequence, occurrences of the nonterminal `finding//0` in a grammar rule's right-hand side are replaced by sequences of `finding, [PT]`, so `PT` holds the parse tree produced by `finding//0`.

In Listing 8.1, we present the adapted DCGs for if-then rules that additionally handle the corresponding parse trees. To keep the compared source code listings minimal, the source code contains only grammar rules and alternatives in their bodies that are required to parse or serialise the example sentence `the weather is rainy.` via the nonterminal `fact//`$\{1, 0\}$. Alternatives as well as logical disjunctions on the right-hand side of grammar rules would be added to this Prolog program by providing additional grammar rules that are extended by the parse tree in a similar way.

Following the graphical representation of the parse tree in Figure 8.1, we omit the handling of whitespaces by the nonterminal `#//0` in our example. Otherwise, line 17 of Listing 8.1 would also contain (a) an additional argument, or (b) a pushback list. The grammar rules for `finding//`$\{1, 0\}$ and `noun_phrase//`$\{1, 0\}$ had to be changed accordingly.

Listing 8.1: Comparison of DCG extensions to process if-then rules with a parse tree

(a) using an additional argument,

```
fact(fact(CT, ".")) -->                    PROLOG
  conjunction(CT), ".".
conjunction(conjunction(FT)) -->
  finding(FT).
finding(finding(FT, ET, VT)) -->
  feature(FT), #, equal(ET), #,
  value(VT).
feature(feature(NT)) -->
  noun_phrase(NT).
equal(equal("is")) --> "is".
value(value("rainy")) --> "rainy".
noun_phrase(noun_phrase(DT, NT)) -->
  determiner(DT), #, noun(NT).
determiner(determiner("the")) -->
  "the".
noun(noun("weather")) --> "weather".
# --> " ".
```

(b) using a pushback list.

```
fact, [fact(CT, ".")] -->                  PROLOG
  conjunction, [CT], ".".
conjunction, [conjunction(FT)] -->
  finding, [FT].
finding, [finding(FT, ET, VT)] -->
  feature, [FT], #, equal, [ET], #,
  value, [VT].
feature, [feature(NT)] -->
  noun_phrase, [NT].
equal, [equal("is")] --> "is".
value, [value("rainy")] --> "rainy".
noun_phrase, [noun_phrase(DT, NT)] -->
  determiner, [DT], #, noun, [NT].
determiner, [determiner("the")] -->
  "the".
noun, [noun("weather")] --> "weather".
# --> " ".
```

Note that Listing 8.1 (b) does not reflect the typical usage of pushback lists. Usually, the state argument is passed around *implicitly* from one grammar rule to the immediately following one, i. e., it is pushed back to the processed list and then read again from the next applied grammar rule. In our example, the state argument on the right-hand side of `finding//0` would therefore be created by processing the nonterminal `feature//0` first, pushed back to the list, to be then consumed by the grammar rule of `equal//0`, which again puts its state back to the list for `value//0`. This approach similarly allows to build the compound term `finding(FT,ET,VT)`. For instance, the parsing process of `finding//0` can be started with the constant `finding` as the initial state argument, and each of the resolved nonterminals `feature//0`, `equal//0`, and `value//0` consumes the term from the processed list, adds its parse tree as additional argument and pushes back the modified compound term, until the Prolog term of functor `finding/3` is finally created. However, since we intend to build a hierarchical term of the applied grammar rules, we refrain from this classical pattern of pushback lists, and follow the less complex approach to directly consume the state argument in the parent rule of `finding//0` instead.

8.2.2. Properties of the Modified DCG

Both modifications of the DCG – processing the parse tree via an additional argument, or via the pushback list – do not alter the core semantics of the grammar rules. Neither non-termination nor nondeterminism is introduced just by the supplementary parse tree processing. In addition, it does not effect the program's performance.

Listing 8.2: Resulting Prolog predicates after source-to-source transformation of `fact//0` and `noun//0` to use pushback lists, and the following DCG term expansion that creates the Prolog predicates `fact/2` and `noun/2`.

```
1 fact(A, [fact(FT,".")|R]) :-                    % ll. 1-2          PROLOG
2   conjunction(A, [FT,'.'|R]).                   %   of Listing 8.1 (b)
3 noun([w,e,a,t,h,e,r|R], [noun("weather")|R]). % l. 16
```

In case of the nonterminal's additional argument on the left-hand side, it is indexed just as another argument of the expanded Prolog clause. The pushback list on the other hand is moved to the clause's head by SWI-Prolog's `dcg_translate_rule/2`$_{\text{SWI}}$ (cf. Section 6.2.4), which also facilitates indexing. For instance, the nonterminal `fact//0` (lines 1–2 of Listing 8.1 (b)) is expanded to a rule with `conjunction/2` in its body, and the parse tree of functor `fact/2` in its second argument. The nonterminal `noun//0` (l. 16) is expanded to a single fact `noun/2`, as the processing of terminals can also be moved to the clause's head. Listing 8.2 shows the expanded clause form of the these two grammar rules. These plain old Prolog clauses for `fact/2` and `noun/2` can again be looked up fast with the help of SWI-Prolog's deep indexing capabilities, so both approaches to modify the DCG are similar regarding their application performance and compared to those of the original, unmodified DCG.

8.2.3. Adapted Use of `phrase/3`$_{\text{ISO}}$

To use these modified DCGs, only calls for `?- phrase(Body, List, Rest)` have to be slightly modified to refer to the parse tree either (a) as an argument to the given nonterminal `Body`, or as (b) the first list element of `Rest`. Therefore, both applications of the modified DCGs in the toplevel presented in Listing 8.3 are equivalent. Each produces the parse tree for the sentence `the weather is rainy.` based on the DCG for if-then rules.

In the computed answer substitution for Listing 8.3, the variable `PT` holds the parse tree we illustrated in Figure 8.1 as a compound Prolog term of functor `fact/1`. Because the right-hand side of the grammar rule for `finding//`$\{1, 0\}$ contains three nonterminals (omitting the whitespace in `#//0`), it is represented by a term `finding/3` in the parse tree. As before, the modified DCGs can be used for both parsing and serialisation, i. e., to create a parse tree by a given source code as in the previous example in the toplevel, and vice versa by providing a bound term `PT` for the parse tree instead. In both directions, only a single solution is returned, as there is exactly one successful SLD resolution to parse the given sentence, or to serialise it respectively.

Listing 8.3: Prolog goals to produce the parse tree for the example sentence `the weather is rainy`, either as (a) an additional argument, or (b) as the pushed back remainder of the difference list.

```
?- phrase(fact(PT), "the weather is rainy.", []).   % (a)
%  phrase(fact,     "the weather is rainy.", [PT]). % (b)
PT = fact(conjunction(finding(feature(
            noun_phrase(determiner("the"), noun("weather"))),
            equal("is"), value("rainy"))), ").
```

TOPLEVEL

8.3. Source-to-Source Transformation

The construction method of the modified grammar rules is generic for both approaches, because the additional parse tree argument and pushback list is constructed based on the rule's left-hand side with its nonterminal symbol and arguments, together with the structure of the rule's right-hand side. Both parts can be examined by Prolog's means for reflection and term inspection. The extension of an existing DCG to additionally work on a corresponding parse tree can therefore be done automatically at compile-time using a generic term expansion scheme.

In the following as well as in our *library(dcg4pt)*, we use the first introduced technique to automatically process the corresponding parse tree when parsing and serialising: it is added as a last argument to the nonterminals. As seen before, the presented considerations can though be easily adapted to the alternative approach using the DCG's semicontext notation.

8.3.1. The Library `dcg4pt`

As part of our contribution, we provide the SWI-Prolog package *library(dcg4pt)*, which is an acronym for "definite clause grammars for parse trees". The library was initially published as *extended DCGs* with the name *library(edcgs)*. In order to avoid confusion with Peter Van Roy's EDCG package, which we elaborate on as related work in Section 8.5, our package has been renamed to the more unique yet descriptive name *library(dcg4pt)*.

The package defines a predicate `dcg4pt_rule_to_dcg_rule(+DCG,-Expansion)` that takes a DCG grammar rule as its first argument `DCG` and returns in `Expansion` a functionally equivalent DCG where the nonterminals have been extended by an additional parse tree argument. The library is listed in SWI-Prolog's package list, therefore it can be conveniently installed by calling `?- pack_install(dcg4pt)` in the toplevel. The package's source code is published under MIT License at

https://github.com/fnogatz/dcg4pt. The correctness of *library(dcg4pt)* is ensured by currently more than 70 unit tests that are specified with and executed by our *library(tap)*, which we previously presented in Section 5.2.2. Its compatibility with new as well as former releases of SWI-Prolog is continuously tested by our version manager *swivm* (cf. Section 1.5.2).

Typically, the source-to-source DCG transformation provided by *library(dcg4pt)* is used to get the extended version of every DCG rule at first. The result is translated afterwards using SWI-Prolog's built-in predicate `dcg_translate_rule/2`$_{\text{SWI}}$:

```
:- use_module(library(dcg4pt)).
term_expansion(H --> B, Rule) :-
  dcg4pt_rule_to_dcg_rule(H --> B, DCG),
  dcg_translate_rule(DCG, Rule).
```
PROLOG

In most cases, one intends to automatically expand all given DCGs in a Prolog source code file into equivalent DCGs with the additional parse tree argument. To do so, *library(dcg4pt)* provides the sub-module *library(dcg4pt/expand)*. Once loaded via `?- use_module(library(dcg4pt/expand))`, all following DCGs are replaced with their extended variant, without the need to manually adjust any grammar rule. Usage examples for this automatic expansion can be found in the library's directory `test`.

8.3.2. Formation Principles

Our *library(dcg4pt)* works as a source-to-source transformation, which takes a DCG and defines the corresponding DCG that additionally processes the parse tree. In contrast to the alternative of modifying the standard term expansion scheme for DCGs (cf. Section 6.2.4), which would directly expand the given DCGs to plain old Prolog clauses that additionally process the parse tree, we can still make use of SWI-Prolog's predicate `dcg_translate_rule/2`$_{\text{SWI}}$ this way. It allows to benefit from the usual optimisations performed for DCGs as part of SWI-Prolog's four preprocessing steps at compilation (cf. Section 5.2). At the same time, the source-to-source transformed DCGs created by *library(dcg4pt)* can be further processed via another term expansion just like regular definite clause grammars.

The general formation principles of *library(dcg4pt)* are presented in Table 8.1. It lists all possibilities that can occur in the body of a DCG rule, together with their modified versions. In addition to terminals and nonterminals, the handling of control structures for conjunctions and disjunctions have to be defined. Furthermore, as similarly done for the DCG meta-interpreter and its the standard term expansion

Table 8.1: Formation principles to construct the parse tree for a DCG rule `h --> Body`.

Body	**Example DCG**	**Extended by Parse Tree**

Terminal	`h --> "_".`
	`h --> [t].`
	`h --> [t,s].`
	`h --> [].`
		`h(h('_'))  --> "_".`
		`h(h(t))  --> [t].`
		`h(h([t,s])) --> [t,s].`
		`h(h([]))  --> [].`

	The inner part of the parse tree is the (possibly empty) list, except in case of a single nonterminal, which is directly used instead.	
Nonterminal	`h --> a.`	`h(h(V)) --> a(V).`

Conjunction	`h --> a , b.`	`h(h(V0)) -->`
		`  { V0 = [A|V1] }, a(A),`
		`  { V1 = [B] }, b(B).`

Disjunction	`h --> a | b.`
	`h --> a ; b.`
		`h(h(V)) -->`
		`  { V = A }, a(A) ;`
		`  { V = B }, b(B).`

Embedded Prolog, and Cut	`h --> a, { p }.` `h --> a, !.`	`h(h(V)) --> a(V), { p }.` `h(h(V)) --> a(V), !.`
	Embedded Prolog and the application of the cut has no effect on the structure of the processed parse tree term.	
Negation	`h --> \+ a.`	`h(_) --> \+ a(_).`
	Negation-as-failure does not bind anything.	
Sequence or Optional (Section 8.4)	`h --> sequence(?, a).` `h --> ?a.`	`h(h(V)) -->` `  sequence(?, a, V).`
	And similar for the prefix operators `*/1`, `**/1`, and `+/1`. *`V` is a list of parse trees.* *In `sequence//3`, we differentiate whether the DCG is called with bound or free arguments.*	

scheme (cf. Appendices B.4 and B.5), it is required to define special cases for Prolog code that is embedded using curly brackets `{...}`, the cut $!/0_{\text{ISO}}$, and negation via $\backslash+/1_{\text{ISO}}$. Without, both last-mentioned would also be recognised as nonterminals and therefore be transformed into `!//1` and `\+//2`, though both do not process a parse tree.

As discussed before, the parse tree is always added as the very last argument to a nonterminal. Though we omit additional arguments in the grammar rule's nonterminals for the sake of simplicity in this overview, they are supported as well, and simply have to be reflected for the nonterminals in the produced rule. For instance, for a rule's left-hand side of $\texttt{h(Arg}_1\texttt{, Arg}_2\texttt{)}$, the head of the generated DCG rule

Listing 8.4: Source-to-source transformation for DCGs with parse trees.

```prolog
%% dcg4pt_rule_to_dcg_rule(+OriginalDCG, -TransformedDCG)
dcg4pt_rule_to_dcg_rule(X1 --> Y1, X2 --> Y2) :-
  X1 =.. [H|_],
  Res =.. [H, V],
  term_args_attached(X1, [Res], X2),
  dcg4pt_formula_to_dcg_formula(Y1, Y2, V).
```

becomes `h(Arg1, Arg2, h(...))`, and likewise for nonterminals that appear in a grammar rule's body.

The left-hand sides of the extended DCGs given in the overview of Table 8.1 are always of the same kind. It is the original nonterminal amended by the parse tree as its new last argument, which again shares a consistent structure of `h(V)`, with only varying bindings for the variable `V`. Consequently, the definition of the predicate `dcg4pt_rule_to_dcg_rule/2` that transforms a given DCG `X1 --> Y1` is split into two parts, as shown in Listing 8.4. In lines 3–5, the variable `Res` that holds the complete parse tree binds to the compound term `H(V)`, which is then added to the list of arguments on the left-hand side `X2` of the modified DCG `X2 --> Y2` via `term_args_attached/3`, which is defined in Appendix C.4. Finally in line 6, the modified grammar rule's right-hand side `Y2` is created based on the formation principles we introduced before.

8.3.3. Modified DCG Body

The source-to-source transformation for DCGs of Listing 8.4 closely resembles the standard term expansion scheme to translate DCGs into plain old Prolog clauses (cf. Section 6.2.4 and Appendix B.5). Both program transformations have in common that additional arguments have to be added to the original terms, and the modifications and definition by cases relate only to the right-hand sides of the processed (grammar) rules, so it can be put into the separate predicates `dcg4pt_formula_to_dcg_formula/3` and `translate_body/4`[B.5]. In both predicates, the first argument takes the original grammar rule that produces a corresponding modified grammar rule or Prolog clause, which then binds the variable stated in its second argument. The `translate_body/4`[B.5] additionally provides access to the processed difference list in the form of two additional arguments. This is required to incorporate the list and its remainder in the produced Prolog clause.

The two arguments for the difference list are not required in the definition of `dcg4pt_formula_to_dcg_formula/3`, as our source-to-source transformation solely

Listing 8.5: Extract of the predicate `dcg4pt_formula_to_dcg_formula/3` that defines the transformation for DCG bodies to handle parse trees. Its full definition is given in Appendix B.8.

```
%% dcg4pt_formula_to_dcg_formula(+Y1, -Y2, ?Value)                    PROLOG
dcg4pt_formula_to_dcg_formula([Terminal], [Terminal], Terminal).
dcg4pt_formula_to_dcg_formula(Terminals, Terminals, Terminals) :-
  is_list(Terminals).

dcg4pt_formula_to_dcg_formula(X1, X2, V) :-
  callable(X1),
  term_args_attached(X1, [V], X2).                                    [C.4]

dcg4pt_formula_to_dcg_formula(Y1, Y2, V) :-
  Y1 = (_,_), !,
  term_functors_list(Y1, [(,)], Ys1),                                 [C.3]
  maplist(conj_body, Ys1, Ys2, V0s, V1s),                             [B.8]
  V0s = [V|V0s_],
  append(V1s_, [Last], V1s),
  Last = [],
  maplist((=), V0s_, V1s_),
  term_functors_list(Y2, [(,)], Ys2).                                 [C.3]
```

works on DCGs in the first place. The handling of the difference list is added in a following step once the modified grammar rule is finally expanded into a Prolog clause. However, the predicate `dcg4pt_formula_to_dcg_formula/3` that is referred to in Listing 8.4 similarly requires access to the inner part `V` of the parse tree `H(V)` from within the transformed grammar rule, which is therefore passed as its third argument. Only after the DCG is executed at run-time, the variable `V` gets bound to either an atom (in case of a single terminal in the grammar rule's right-hand side), a compound term (nonterminal), or a list (multiple terminals or a sequence). Therefore, unification of `V` happens via embedded Prolog code enclosed in curly brackets `{...}` in the produced grammar rule, as depicted in the modified DCGs for, e. g., conjunctions and disjunctions in Table 8.1.

Listing 8.5 shows the definition of the predicate `dcg4pt_formula_to_dcg_formula/3` for the first three rows of formation principles we introduced in Table 8.1. The first part processes terminals, which are either a single terminal in a list with only one element, to which the inner parse tree `V` binds to (l. 2), or a list of terminals (ll. 3–4). For a single nonterminal, the variable `V` binds to the parse tree generated for it. It can be accessed in the modified grammar as the newly introduced last argument of the original callable `X1` (ll. 6–8).

The modifications required for conjunctions in DCG are the most complex and given in lines 10–18. The conjunctions are first transformed into lists and afterwards back to compound terms of functor `,/2` with the predicate `term_functors_list/3`. The inner part of the clause creates the modified DCG body and binds the variables accordingly. In line 13, each of the body elements in `Y1` is transformed into a modified version that handles its parse tree. The predicate `conj_body/4` creates the DCG body for a single nonterminal following the formation principle introduced in Table 8.1, with two chaining variables for the processed difference list. For instance, for a single nonterminal `a`, the modified DCG body `Y2` is returned as follows:

```
?- dcg4pt:conj_body(a, Y2, V0, V1).                                    Toplevel
Y2 = ({V0 = [V_a|V1]}, a(V_a)).
```

The variable `V_a` binds at run-time to the parse tree of the nonterminal `a`. The following lines 14–17 pairwise unify the chaining variables introduced by the multiple calls of `conj_body/4`, starting with $\texttt{V = V0}_1$ and $\texttt{V0}_1\texttt{ = V1}_0$, over $\texttt{V}k_1\texttt{ = V}(k+1)_0$ for the k-th DCG body, and concluding with the empty list $\texttt{V}n_1\texttt{ = []}$ for the very last body element. This binding for $\texttt{V}n_1$ closes the list that represents the overall inner value `V`.

The predicate `conj_body/4` internally refers to `dcg4pt_formula_to_dcg_formula/3` again to transform the nested DCG body elements, and additionally handles sequences. The full definition of both predicates is given in Appendix B.8, including the other cases that were given in Table 8.1.

8.4. Optionals and Sequences of Nonterminals

Grammars that describe formal languages often make great use of optional nonterminals as well as sequences thereof. For instance, for computer languages whose syntax is not whitespace-sensitive (e. g., Prolog), the tokens can be separated by any positive number of spaces, tabs, and newline characters for indentation, and can include optional source code annotations. In addition, the format of these tokens is usually described as a sequence of allowed characters. In this way, an integer constant is built by a sequence of decimal digit characters; an uppercase letter character followed by an arbitrary number of alphanumeric characters forms a valid Prolog variable symbol. In this section, we discuss the means to handle a variadic number of processed terminals and their implementation in *library(dcg4pt)*.

8.4.1. Parse Trees with Lists

The usage of sequences to an arbitrary number of identical nonterminals differs from the previous applications with parse trees of finite children, which was mainly discussed in the field of natural language processing, e. g., in [2]. There, the number of nonterminals processed in a grammar rule's right-hand side is known in advance. As a result, the parse tree argument can have a fixed number of arguments and children, i. e., the functor of the processed Prolog term can be inferred by static code analysis at compile-time. It is solely based on the grammar rule's structure. For instance, for a fact in if-then rules that is simply a conjunction, the parse tree is represented as the unary structure `fact(Conjunction)`. A finding that consists of a noun phrase with its parse tree as `Feature`, one of the allowed equal phrases `=`, `is`, or `are` as `Equal`, and a parse tree representation for the finding's value as its last argument `Value` is on the other hand represented as `finding(Feature, Equal, Value)`.

When working with optionals and sequences on the other hand, the number of children can change and could be not limited to a fixed value that is known in advance, just by statically analysing the grammar rule. It is therefore desirable to use a list in the parse tree structure if the grammar rule's right-hand side contains at least one optional or one sequence, and rely on a fixed number of arguments otherwise. For a right-hand side which contains only a finite, fixed number of arguments, we could also use a single list instead of separate arguments. But since we observed that in practice the number of grammar rules that process a flexible number of terminals is much less than those with a fixed sequence, we aim for the more natural representation as compound Prolog terms.

For instance, the DCG for a named variable in Prolog should produce a parse tree of the form `named_variable(L)`. The general structure of a named variable has been given in Figure 5.2 in the form of an EBNF, which we present in a shortened version here again:

EBNF

$$
\begin{aligned}
\textit{named variable} \;=\; & \textit{variable indicator char}, \\
& \textit{alphanumeric char}, \\
& \{\ \textit{alphanumeric char}\ \} \\
\mid\; & \textit{capital letter char}, \\
& \{\ \textit{alphanumeric char}\ \}\ ;
\end{aligned}
$$

In the compound term `named_variable(List)` that represents the parse tree of a successful DCG application, the variable `List` either holds a list consisting of the parse tree that represents the variable indicator character, followed by the non-empty

sequence of parse trees which each represent an alphanumeric character. Alternatively, for the second part of the disjunction, `List` holds a parse tree for an uppercase letter character followed by the possibly empty list of parse trees that each represent an alphanumeric character.

8.4.2. Handling and Transformation

To denote optionals and sequences of nonterminals, we introduce the meta-nonterminal `sequence//2` that can be used as a DCG body element. A grammar rule's right-hand side `sequence(?Mode, :NT(Arg`$_1$`, ..., Arg`$_n$`))` is processed as a sequence of the nonterminal *NT*`//`n with arguments `Arg`$_i$. It is expanded by our source-to-source transformation to `sequence(?Mode, :NT(Arg`$_1$`, ..., Arg`$_n$`), ?PTs)`, as previously presented in Table 8.1. The created `sequence//3` additionally holds the sequence's corresponding list of parse trees as the variable `PTs`. This transformation is only a special case of the formation principle introduced for nonterminals. Therefore, it does not need to be explicitly implemented. The original nonterminal `sequence//2` is just amended by the parse tree as the last, third argument, like for other nonterminals.

The variable `Mode` is one of the atoms `*`, `+`, and `?`, which correspond to the symbols for repetitions known from regular expressions. The mode `*` depicts a sequence of an arbitrary number, including zero, i. e., the grammar rule's body element *NT*`//`n could also consume or create the empty list. We additionally provide the mode `**` with the identical meaning but different order in case of backtracking. The mode `+` denotes a sequence with at least one occurrence of *NT*`//`n, and the mode `?` depicts an optional nonterminal, i. e., of exactly zero or one occurrence.

Sequences of modes `*` and `**` both consume an arbitrary number of elements and differ only in their strategy for backtracking: sequences of `*` process as few as possible nonterminals first, while `**` greedily consumes the elements, beginning with the longest possible sequence of the given nonterminal. For unambiguous and terminating DCGs, both variants are semantically equivalent, and differ only in their execution times.

The meta-nonterminal `sequence//2` is translated into `sequence//3` by our source-to-source transformation to additionally process the parse tree. With the standard term expansion scheme for DCGs, this again is translated at compile-time to the Prolog predicate `sequence/5`. It allows to process a DCG in various instantiation modes, i. e., with the parse tree argument or parts of the difference lists as (possibly partially) free variables. For instance, if the repetition mode `Mode` in `sequence/5` is a free variable, it is inferred by the given difference list and/or parse tree. As

Listing 8.6: Computing all answer substitutions for optionals and sequences of the nonterminal `n//0` in the toplevel via backtracking.

```
% DCG grammar rule n --> [t]. produces n(t) parse trees                TOPLEVEL
?- sequence(Mode, n, PTs, [t, t], Rest).
① Mode =    ?, PTs = [n(t)],       Rest = [t] ;
   Mode =    ?, PTs = [],           Rest = [t, t] ;
② Mode =  (*), PTs = [],           Rest = [t, t] ;
   Mode =  (*), PTs = [n(t)],       Rest = [t] ;
   Mode =  (*), PTs = [n(t), n(t)], Rest = [] ;
③ Mode = (**), PTs = [n(t), n(t)], Rest = [] ;
   Mode = (**), PTs = [n(t)],       Rest = [t] ;
   Mode = (**), PTs = [],           Rest = [t, t] ;
④ Mode =  (+), PTs = [n(t)],       Rest = [t] ;
   Mode =  (+), PTs = [n(t), n(t)], Rest = [] .
```

a result, calling `?- sequence(Mode, NT, PTs, In, Rest)` in the instantiation mode `(-,:,-,+,-)` with a known input list `In`, and free variables `Mode` for the mode, `PTs` for the list of parse trees, and `Rest` for the difference list's remainder, computes all valid modes and variable bindings for a given nonterminal `NT`.

Listing 8.6 shows the output produced by `sequence/5` in the toplevel for the minimal exemplary DCG `n --> [t]`, which could be equally written as `n --> "t"` if the Prolog flag `double_quotes` is set to `chars`. This grammar rule describes the nonterminal `n//0` that consumes only the single atom `t`. The parse tree for a non-recurring successful application of `n//0` is simply the compound term `n(t)` according to the representation introduced in Section 8.1. Then, given an input list of `[t, t]`, ten possible answer substitutions are computed. We shortly summarise the backtracked results for each of the four modes:

① For the repetition mode `?`, there are only two computed answer substitutions, since the nonterminal `n//0` is applied only once or never, leaving either one or two elements in the remainder list `Rest`.

② The mode `*` applies the nonterminal `n//0` an arbitrary number of times. Beginning with the empty sequence, the number of consumed list elements increases on backtracking and concludes with an empty remainder list `Rest`.

③ Since the mode `**` instead greedily consumes list elements, its order is the inverse of the previous mode `*`.

④ The repetition mode `+` takes at least one element and subsequently behaves like `*`. Therefore there are two computed answer substitutions with an increasing number of consumed list elements.

Listing 8.7: Transformed grammar rule with a conjunction and sequence.

```
1 % Original: h --> a, sequence(*, b), c                                PROLOG
2 h(h([A,Bs,C])) --> a(A), sequence(*, b, Bs), c(C).   % naive approach
3 h(h(V0)) -->                  % as instead produced by library(dcg4pt)
4   { V0 = [A|V1] }, a(A),
5   sequence(*, b, Bs), { append(Bs, V2, V1) },  % works only for bound
6                                                % difference list
7   { V2 = [C] }, c(C).
```

The implementation of `sequence//3` in the form of a DCG is given in Appendix B.7, together with the declaration of the corresponding Prolog meta-predicate `sequence/5`. The order of the computed answer substitutions as shown in Listing 8.6 is determined by the order of appearance of the grammar rules in the definition of `sequence//3`.

With respect to conjunctions in a grammar rule's body, the variadic form of the corresponding parse tree has to be considered to support complex right-hand sides with all possible combinations of structures and body elements. This is why the formation principles for the modified DCG which were given in Table 8.1 make use of embedded Prolog code snippets for the variable unifications in case of conjunctions. For instance, a naive transformation for the DCG rule `h --> a, sequence(*, b), c` results in the grammar rule given in line 2 of Listing 8.7.

But since `sequence//3` describes a list `Bs`, the generated parse tree for `h//1` would contain a list of lists in its second argument instead of the preferred flattened parse tree representation for the overall conjunction. Consequently, the extended DCG given in lines 3–6 of Listing 8.7 is instead preferred and generated by our *library(dcg4pt)*. If there is no `b//1` in the sequence, the resulting parse tree for `h//1` is just the compound term `h([A, C])`, with `A` and `C` being the parse trees generated for the nonterminals `a//1` and `c//1`. Otherwise, the parse trees generated by the applications of `b//1` are placed in between `A` and `C` in the inner list `V0`.

8.4.3. Support for Parsing and Serialising

The aim of *library(dcg4pt)*'s source-to-source transformation is to create a modified DCG that expresses a relation between the input list and the corresponding parse tree, i. e., from the processed string of the external DSL to some kind of (abstract) syntax tree and vice versa. In particular, the generated Prolog program can also be used "in reverse" to a normal parser to serialise a string by its given parse tree.

For this purpose it has to be ensured that the term expansion scheme presented in Section 8.3 uses only Prolog predicates that are pure (cf. Section 3.2.3), i. e., they can be used no matter which of the arguments are bound. Alternatively, these predicates have to be implemented impure with a detection and differentiation of the executed mode, usually inferred by inspecting the argument's variable bindings via `var/1`$_{\text{ISO}}$ and `nonvar/1`$_{\text{ISO}}$.

For instance, the aforementioned rule `h --> a, sequence(*, b), c` could also be expanded to use SWI-Prolog's built-in predicate `flatten/2`[C.12] instead of `append/3`[3.4]. The predicate `flatten/2` calculates a flattened list from a list of lists and therefore also avoids the use of nested lists in the resulting parse tree. Its full definition is given for reference in Appendix C.12. It emphasises that the goal `?- flatten(+ListOfLists, -FlattenedList)` cannot be executed the other way round, because it is logically impure due to its dependency on the cut `!/0`$_{\text{ISO}}$. As a result, the generated modified DCG can only be used to parse a given string and return the corresponding parse tree; serialisation of a given parse tree back to the corresponding string is not possible with the required instantiation mode `(+,-)`.

In addition, because of possibly left-recursive grammar rules, or rules that consume or produce no terminals, the expanded rules have to behave differently depending on whether they are called with bound or free arguments. For instance, consider the rules that describe a *named variable* as presented in the EBNF from Section 8.4.1: in the first alternative, it is the *variable indicator char* `_`, followed by a non-empty sequence of *alphanumeric char*. For a given string `_abc`, the right-hand side in the EBNF for *named variable* should first consume the single character `_`, followed by as many whitespace characters as possible to avoid unnecessary backtracking – the behaviour that is achieved by the repetition mode `**`. On the other hand, this greedy approach is undesirable when both arguments are free, i. e., when generating all allowed strings that form a valid named variable in Prolog, together with their corresponding parse tree. In that case, the smallest possible string should be created at first – effectively expressing the repetition mode `*` –, as otherwise the query never terminates. In case of the transformed grammar from Listing 8.7, an free difference list of terminals leads to the execution of the subgoal `?- append(Bs, V2, V1)` in the embedded Prolog code in line 5. However, following the SLD resolution and the predicate's definition from Section 3.4, it first backtracks over a growing list in the first argument, leaving `V2` and `V1` unchanged.

It is therefore necessary to handle the four combinations: the processed list can be free our (partially) bound, and the same applies for the parse tree argument. The four different call modes are thus automatically handled by the DCGs generated by our *library(dcg4pt)*. Instead of relying on `append/3`[3.4] as previously shown in

Listing 8.7, we call the *library(dcg4pt)*'s predicate `call_sequence_ground/6`. This way, line 5 becomes `call_sequence_ground(sequence(*, b, Bs), Bs, V2, V1)` instead.

The definition of the predicate `call_sequence_ground/6` is given in Listing 8.8. Based on the known bindings of the variable `V0`, which depicts the overall parse tree, and the processed difference list `A-Z`, we either first reduce the known parse tree to smaller parts (ll. 3–6), or the list of terminals `A` (ll. 7–11). Calls to the predicate are inserted in the modified DCG body as part of the source-to-source transformation using `conj_body/4`, as shown in the predicate's definition in Appendix B.8. Besides being bound (l. 8), the difference list `A-Z` could also be a free *attributed variable*, which is checked via `attvar/1`$_{\texttt{SWI}}$ (l. 9). This concept, which we introduce in more detail in Section 10.1.3, is used in SWI-Prolog to lazily read the input from an external source into the processed list on demand, constituting an open difference list. Therefore `nonvar/1`$_{\texttt{ISO}}$ fails, but SWI-Prolog's implementation of `phrase/3`$_{\texttt{ISO}}$ handles the attributed variables `A` and `Z` as expected.

If all of the arguments of `call_sequence_ground/6` are free, our *library(dcg4pt)* prints a warning. Though the modified grammar can be used to generate all allowed lists with their corresponding parse trees, it most likely will not end with the expected result, like previously mentioned in this chapter's introduction. This is because of the underlying SLD resolution, which backtracks the last possible alternative first. In our example of named Prolog variables, it results in the non-terminating sequence of growing variable names `_a`, `_aa`, `_aaa` instead of the possibly more intended and descriptive sequence that iterates over the allowed *alphanumeric char*, resulting in the sequence of `_a`, `_b`, and so on. With both arguments being free variables, `phrase/3`$_{\texttt{ISO}}$ first backtracks over the sequence, not the symbols. Changing this requires a separate meta-interpreter for the execution of the DCG. The meta-interpreter for definite clause grammars which we presented in Section 6.2.3 could serve as a basis for that sort of adaptions.

Similar to the definition of the predicate `call_sequence_ground/6`, checks for free and bound variables have been implemented for the application of sequences using the `sequence//3` nonterminal. The resulting Prolog programs can be used both for parsing and serialisation, based only a single grammar specified in the form of a DCG.

Listing 8.8: Changing the order of body elements in the transformed DCG depending on whether the parse tree argument is bound or the difference list.

```
%% call_sequence_ground(DCGBody, Ref, Rest, PTs, A, Z)                    PROLOG
:- meta_predicate call_sequence_ground(//, ?, ?, ?, ?, ?).
call_sequence_ground(DCGBody, V, V1, V0, A, Z) :-
  nonvar(V0), !, % parse tree bound
  append(V, V1, V0),
  phrase(DCGBody, A, Z).
call_sequence_ground(DCGBody, V, V1, V0, A, Z) :-
  ( nonvar(A) % difference list bound
  ; attvar(A), get_attr(A, pure_input, _PIO)), !, % or lazy list
  phrase(DCGBody, A, Z),
  append(V, V1, V0).
```

8.5. Related Extensions for DCGs

Natural Language Processing. Since its introduction by Colmerauer, Prolog was developed with a focus on natural language processing. This resulted in a first representation of grammars as clauses of first-order logic in 1975 by Colmerauer [21, 22]. Definite clause grammars were introduced by Pereira and Warren in 1980 [96]. As a usage example of extra arguments in nonterminals, they manually extend rules that parse sentences by their corresponding *building structures* – a term holding information about the applied rule and the elements of the rule's right-hand side. To support the linguistic phenomena known as "left extra-position", *extraposition grammars* have been introduced as an extension to DCGs [95].

This idea of creating terms about the applied grammar rules was adopted by Dahl and McCord in 1983 [30]. Their *modifier structure grammars* extend a grammar for sentences in natural language with two additional arguments to obtain a meaning representation (called *semantic structure*) and its corresponding *syntactic structure* in the form of a parse tree. Simultaneously and independently, *restriction grammars* were developed by Hirschman and Puder [51]. Their work also contains automatically created parse trees. With *definite clause translation grammars* (DCTGs) [1], it is possible to handle grammatical attributes conveniently. The translation of DCTGs into normal Prolog clauses is like that of DCGs, but a third argument is added that holds a computed attribute of the node. An overview of these three approaches is given in [2, Chapters 7–8], where the idea of hiding the parse tree argument from the user is discussed.

The aforementioned approaches are focussed on context-free grammars, covering only the expressiveness of EBNF. In particular, they do not make use of embedded

Prolog code snippets on a rule's right-hand side, and higher-order structures like sequences. Although they expand grammar rules by an additional argument to store a parse tree, its actual construction in a generic expansion scheme is not specified. Hence, we have observed that they do not address the challenges that arise when grammar rules, that consume or produce no symbols, are called with free variables as arguments. This is a requirement for grammars which are to be used for both parsing and serialisation.

Extended DCG Notation by Van Roy. Peter Van Roy introduced the *extended DCG notation*, or EDCG for short, in [122]. It was published in 1990 as part of the *Aquarius Prolog* system [121, 124].The original Prolog source code of EDCGs as implemented by Van Roy is archived at https://github.com/mndrix/edcg. A maintained version is available from https://github.com/kamahen/edcg (both MIT License). In the latter repository, Van Roy's original code has been adapted and improved by Peter Ludemann to work as an add-on for SWI-Prolog, so it can be conveniently installed by calling `?- pack_install(edcg)` in the toplevel.

EDCGs treat the difference list that is processed by definite clause grammars as an *accumulator*, i. e., two variables that are chained together by an accumulator function so they can be used for parameter passing. While DCGs provide a single, hidden accumulator in the form of a difference list in its last two arguments, EDCGs allow to define Prolog predicates that have arbitrarily many hidden accumulators. The additional arguments can be used to define multiple accumulators of any type, e. g., to calculate and store the size of the consumed terminal symbols. The hidden arguments need to be declared ahead of the grammar's first call using the library's predicates.

Thus, it is feasible to use EDCGs with a hidden accumulator that creates the corresponding parse tree, similar to our *library(dcg4pt)*. To the best of our knowledge, this has not yet been done, as there is currently no application that uses EDCGs with an accumulator that creates a nested compound Prolog term of functors based only on the processed nonterminal symbols. This might be due to the fact that though user-defined accumulator functions are declared just as normal Prolog predicates, they only provide access to the former and following value of the accumulator (e. g., in case of difference lists the splitting in the head and remainder), as well as to a passed Prolog term. Context information about the currently applied EDCG rule is nevertheless missing in the accumulator function. Similar to the term expansion for traditional DCGs, we therefore require a preceding program transformation to augment the EDCG rules by the needed information in order to avoid describing the corresponding parse tree's structure manually in the rule's right-hand side.

Nevertheless, in practice we often encountered use cases where it would be useful to carry more than one implicit accumulator in the DCGs generated by *library(dcg4pt)*, for instance in the passing of program-wide flags and options in our *library(plammar)*. With classical DCGs, these parameters have to be provided by additional arguments to the nonterminals in the rule's left-hand side, which usually requires the invention of many arbitrary variable names, and the chance of introducing errors is large. In addition, modifying or extending an existing definite clause grammar by an additional argument, is tedious. With EDCGs, it requires the definition of just another accumulator, without further modifications to the underlying grammar.

Extended DCGs by Seipel et al. Our *library(dcg4pt)* has its origins in the *library(edcg)* module provided as part of the *Declare Developers' Toolkit DDK* package [103]. This library has been used in an application of DCGs to language processing for electronic dictionaries in linguistics by Seipel et al. [102], where parse trees are represented as an XML term in Prolog. It is available for download at https://www1.pub.informatik.uni-wuerzburg.de/databases/ddbase/.

The *library(dcg4pt)* is an improvement on the previous implementations of parse tree handling in DCGs of [30] and [102]. Though it evolved from the DDK's *library(edcg)*, it is more focussed on the applicability for both parsing and serialisation, and to work as a drop-in replacement for Prolog's standard term expansion scheme for DCGs. In addition, it adds support for the four repetition modes `?`, `*`, `**`, and `+` of optionals and sequences, and dynamically decides to either reduce the difference list or parse tree into smaller parts first, depending only on the applied instantiation mode.

9

A Prolog Parser and Serialiser in Prolog

Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

— Greenspun's tenth rule of programming

Philip Greenspun's tenth rule of programming (circa 1993) expresses the opinion that the flexibility and extensibility of complex systems implemented in low-level languages require the same facilities as needed to implement a subset of the methods used in Lisp. Originally specified in 1958 – a year after Fortran –, Lisp (historically "LISP") is the second-oldest high-level programming language, though it is still in widespread use today. Its best-known general-purpose dialects are *Racket*, *Common Lisp*, *Scheme*, and *Clojure*. Lisp's characteristic, fully parenthesised prefix notation makes no distinction between expressions and statements, i. e., both code and data is written as expressions that are evaluated. Since Lisp functions are notated as lists, they can be processed exactly like any other data, which allows to reason about programs and manipulate them via meta-programming, similar to Prolog.

Greenspun's tenth rule can also be interpreted as a satirical critique of systems that include complex, highly configurable sub-systems, often leading to large, hardly maintainable software projects with data exchange formats and interfaces to several external systems. Rather than including a custom interpreter for some domain-specific language in a high-level programming language, Greenspun's tenth rule suggests using a widely accepted, fully-featured language with an extensible syntax instead – like Lisp. In this regard, it emphasises the use of a general-purpose language with a flexible syntax that allows the definition of internal domain-specific languages as opposed to complex systems of a higher-level language that require parsers and interpreters.

Prolog fits well into these considerations. With its flexible syntax, it allows to embed many domain-specific languages internally. Together with its homoiconicity property, this allows to define the DSL in the same language as it is further processed. By using Prolog's term expansion capabilities, writing a dedicated meta-interpreter,

or by just querying the thus obtained Prolog database as-is, there is no need for a dedicated parser. As a consequence, the integration of languages into Prolog as internal DSLs has been proven useful particularly for rapid prototyping. With the various operators that are defined in the ISO Prolog standard and that are shipped by the used Prolog system, many common constructs like arithmetic expressions (using, e. g., `+/1`$_{\texttt{ISO}}$, `+/2`$_{\texttt{ISO}}$, `*/2`$_{\texttt{ISO}}$) and assignments via `=/2`$_{\texttt{ISO}}$ can be stated verbatim without further modifications. For more expressive knowledge bases, usually only a small number of user-defined operators has to be declared. This consequently leads to an easily maintainable yet powerful system to integrate and store knowledge of any application domain.

In this chapter, we present the foundations of our *library(plammar)*, which assists with the process of defining a DSL internally in Prolog. We adapt the concepts of Chapters 6 and 8 and treat Prolog as if it were an external domain-specific language, i. e., we parse it with the help of Prolog and definite clause grammars. As a result, possible operator definitions and required language extensions that cause the given sentence to be valid Prolog are returned. For this purpose, *library(plammar)* makes use of the EBNF given in the ISO Prolog standard that formally describe the syntax of Prolog, as well as of our *library(dcg4pt)* to create the corresponding parse trees. The result is a flexible and deducing Prolog parser, completely written in Prolog, which can be equally used the other way round as a Prolog serialiser. Consequently, it also supports various other applications like static code analysis, and complex program transformations.

We introduce *library(plammar)* in more detail in Section 9.1. In Sections 9.2 and 9.3, we describe its first phase of lexical analysis, which converts a sequence of characters into a list of tokens. The second phase of parsing the tokens and combining them in order to generate a structural and meaningful representation as Prolog terms is described in Section 9.4. Quite often, applications do not rely on all source code layout information. Therefore, we present in Section 9.5 how to transform this concrete into an abstract syntax tree. The description of the constraint satisfaction problem that is formed by the operators' precedences and associativities follows in Chapter 10. In both chapters, we frequently rely on the exact syntax definitions as given in *Part I* of the ISO Prolog standard. For better readability, we refer to them using the abbreviation Iso, with Iso 6.3 referencing Section 6.3 of [55].

9.1. The Library `plammar`

As part of our contribution, we provide the SWI-Prolog package *library(plammar)* – a Prolog grammar written in Prolog, for parsing and serialising Prolog code. The

word “plammar” is a portmanteau for “Prolog grammar”. The library is published under MIT License at https://github.com/fnogatz/plammar. It is available for installation from SWI-Prolog’s list of add-ons, therefore it can be conveniently installed by calling `?- pack_install(plammar)`. It depends on our *library(tap)* for the definition and application of input/output tests, *library(dcg4pt)* for the automatic parse tree extension for DCGs, and our *library(cli_table)* to print nicely formatted ASCII tables in *library(plammar)*’s command line interface. The correctness of *library(plammar)* is ensured by currently more than 300 unit tests that are specified with and executed by our *library(tap)* (cf. Section 5.2.2). In addition, it is used in [82] to analyse real-world Prolog applications. For this empirical study, about 4500 Prolog files from 280 packages shipped with SWI-Prolog and its add-ons are parsed in order to check their adherence to common Prolog coding guidelines.

In this section, we first motivate in Section 9.1.1 further applications for our *library(plammar)*, and how they profit from a fully-featured Prolog parser written in Prolog. On top of the suggested use case to infer operators for possible DSL integrations, *library(plammar)* can be used to analyse Prolog source code files, and to define source-to-source transformations. In Section 9.1.2, we present a summary of the predicates that are provided by *library(plammar)*. Section 9.1.3 describes its standalone command line interface. The section concludes in Section 9.1.4 with an overview of the package’s underlying concepts and its tokenisation and parser components.

9.1.1. Intended Applications

All the advantages of using Prolog to model expert knowledge in a particular application domain using an internal DSL apply only if the process of defining the required Prolog operators is fairly easy. Currently, this process is often driven by incrementally improving a small, internal Prolog DSL, starting from only the built-in operators. However, in an environment focussed on rapid prototyping, it is often intended to start by expressing the overall problem, and divide it into smaller parts. *library(plammar)* supports this process by automatically detecting feasible operator definitions that make a set of given example sentences valid Prolog programs.

However, a fully-featured Prolog parser written in Prolog not only allows to solve the constraint satisfaction problem constituted by missing operator definitions. Even with all precedences and associativities known, there are various applications for the parser that benefit from the fact that *library(plammar)* is completely written in Prolog. In the following, we shortly introduce some of these applications.

Static Source Code Analysis. Tools for static source code analysis have been established in the process of software engineering in all major programming languages. They allow developers to discover potential faults early. For instance, a misspelled variable name is never used again, and therefore throws a singleton warning. Besides the prevention of software bugs, static code analysis could highlight means for optimisations, e. g., to eliminate loop invariants.

One way to identify so-called *code smells* is by visualising the source code, i. e., to illustrate modules, predicates, and their connections. Since *library(plammar)* parses Prolog like an external DSL, it also constructs a corresponding parse tree on-the-fly, which can serve as a starting point to create source code visualisations, and to find duplicated code or strongly coupled predicates. Another useful property is the usage of programming patterns that go beyond the ISO Prolog standard. Using static source code analysis, a given Prolog program can be tested for compatibility with different target environments, i. e., in which Prolog systems it can be used without or with only minor changes, or if a particular version thereof is required.

Enforcement of Coding Conventions. Another typical static source code analysis is regarding its adherence to coding standards. This is useful in particular for open-source projects with a wide variety of contributors with different backgrounds. In combination with a continuous integration (CI) pipeline, this allows to automatically enforce the project's naming conventions and consistent code indentations [142]. In our work [82], *library(plammar)* has been used to empirically evaluate modules that are shipped with SWI-Prolog or provided in its list of add-ons regarding their adherence to some popular Prolog coding guidelines of [25].

For instance, a typical coding standard is to limit the length per source code line to a fixed number. Covington et al. suggest to use a column width of 78 or 79 characters, which can be lowered for maximum readability down to 55 characters. Since Prolog has no distinction of variables for input and output, in the linter based on *library(plammar)* we can simply call the coding convention setting `max_line_length(Int)` with a free variable, as it will bind `Int` to the maximum column width of the examined Prolog file. This neat application of logic variables emphasises the use of Prolog for implementing linters and static source code analysis tools for software projects written in Prolog, or any other language.

Code Formatter. While linters usually just warn about the breach of coding conventions, code formatters are used to automatically rewrite source code in the desired way, which ultimately results in a program transformation. These typically require to first parse the program based on a grammar, then generate an abstract syntax

tree, modify it, and finally serialise it again. As introduced with our *library(dcg4pt)*, we can use the same language specification – i. e., the same grammar and code – for the parsing and serialisation steps. They share a single data structure: the parse tree which was automatically added by our tool's modified term expansion. The resulting Prolog program can be used in both directions without any modification. Compared to the common approach to use a parser generator like ANTLR [93] instead, *library(plammar)* relieves the programmer from the burden of keeping two tools, for parsing and serialisation, in sync.

Incremental Rollout of Experimental Language Features. Besides operators, some DSLs require minor syntactic changes to Prolog to be valid and meaningful terms, like the `var_prefix`[D.7] program flag. Though the fact `Prolog is great` is a syntactically valid if-then rule, it becomes relevant only if `Prolog` is (in contrast to the ISO Prolog standard) not handled as a variable but atom, as otherwise the universally quantified variable `Prolog` implicates that *everything* is great. By setting `var_prefix(true)`, the fact can be used as-is, which underlines the usefulness of such program flags for the expressiveness of Prolog, though they are supported by only a minor part of the Prolog systems, and are not standardised at all.

In other programming languages, it has proven to be useful to have implementations that are flexible enough to enable and disable such individual language features, for backwards compatibility on the one hand, but also for testing new, experimental syntactic changes before going productive on the other. For instance, new language features that are discussed for the inclusion in JavaScript go trough five stages (strawperson – proposal – draft – candidate – finished), before being officially released in the ECMAScript standard, thus evolving an addition from an idea to a fully specified feature, complete with acceptance tests and implementations in multiple systems. In this process, newly discussed features are available for usage, tests, and real-world applications from the beginning. Even if not yet implemented in all systems, their adoption is facilitated by the availability of program transpilers that polyfill features which are missing in the target environment by source-to-source transformations to backwards compatible standard expressions. In the case of JavaScript, which experienced the addition of far-reaching changes and significant new syntax for writing complex applications when moving to ECMAScript 2015 (formerly ECMAScript 6) [141], the open-source transpiler *Babel*[34] became the de-facto standard to support legacy browsers, with more than 16 million downloads per week in 2019.[35]

[34] Babel JavaScript compiler, https://babeljs.io/, MIT License.
[35] Blog post "Babel's Funding Plans" (November 2019), https://babeljs.io/blog/2019/11/08/babels-funding-plans.

A similar toolchain to incrementally implement, test, and adopt new language features is in contrast missing in the Prolog community, even though a common platform is more tempting with the larger number of different systems that implement the ISO Prolog standard or dialects thereof, compared to the manageable number of JavaScript platforms. After all, many (all?) Prolog systems diverge from the ISO Prolog standard.[36] Their parsers have bugs, are written by people who misunderstand the specification, or are extended by proprietary syntax to introduce new language features [82]. With *library(plammar)*, we provide a unified platform that targets only the Prolog syntax, without an associated run-time. It can help to identify incompatibilities between existing systems and Prolog applications, as well as a foundation to incrementally define and test new language extensions. A major advantage compared to its implementation in other languages is *library(plammar)*'s used technology stack: based only on Prolog and DCGs, it is easily understandable for the targeted audience of Prolog system implementers and users.

9.1.2. Provided Predicates

Analysing the syntax of a programming language usually requires two phases: (i) the lexical analysis, that converts a sequence of characters into a sequence of tokens, and (ii) the parsing of the tokens in order to generate a structural representation. Our *library(plammar)* similarly provides the two predicates `prolog_tokens/`$\{2,3\}$ and `prolog_parsetree/`$\{2,3\}$ which convert some Prolog source code into a list of tokens or into the parse tree. The first always succeeds, as any input can be lexically analysed. The predicate `prolog_parsetree/`$\{2,3\}$ on the other hand only succeeds for a valid Prolog program. In this case, it returns its parse tree, which is the program's concrete syntax tree (CST), i. e., including all source code information like line breaks, whitespaces, and annotations. This predicate implicitly performs the lexical analysis via `prolog_tokens/`$\{2,3\}$ first; however, our module exports no public predicate to manually convert a known list of tokens to the corresponding parse tree.

Both predicates expect the Prolog source code as their first argument, and the representation as token list or parse tree as the second. The third optional argument can be used to provide a list of settings that can be processed by SWI-Prolog's *library(option)* [136, Sec. A.26]. If not specified, the empty list is used, thus falling back to the settings' (reasonably chosen) default values. The options are typically specified in the format `Property(Value)`. Most allow only a Boolean `Value`, which

[36] *Conformity Testing I: Syntax*, list of compliance to the ISO Prolog standard for popular Prolog systems, collected by Ulrich Neumerkel: https://www.complang.tuwien.ac.at/ulrich/iso-prolog/conformity_testing.

is stated as `true` or `false`. For compatibility with *library(option)* and SWI-Prolog's program flags, we also support the atoms `yes` and `no`. The properties are formulated so that the implicit default is always `false`.

In many applications, only the program's abstract syntax tree (AST) is of interest, for instance in case of compilation or interpretation. The *library(plammar)* therefore also ships with the predicate `prolog_ast/`$\{2,3\}$, which removes from the parse tree all information that is not necessary for the depicted program's execution, i. e., all contained source code annotations and layout information.

Following the ideas of *library(dcg4pt)*, all predicates provided by *library(plammar)* have been developed with a focus on their usage as relations, i. e., with support for various instantiation modes. Thus they handle the input of only the Prolog source code in their first argument, or only the tokens, parse tree, or abstract syntax tree in their second argument. We also support queries with all arguments being free variables, including the list of options in their third argument.

All of the six predicates `prolog_tokens/`$\{2,3\}$, `prolog_parsetree/`$\{2,3\}$, and `prolog_ast/`$\{2,3\}$ support multiple formats to represent the processed Prolog source code in their first argument:

- `stream`: `stream(user_input)`
  The input is read from the given stream identifier, which is either explicitly created using the predicate `open/3`$_{\text{ISO}}$, or a built-in alias like `user_input`.

- `file`: `file('~/append.pl')`
  The input is read from the file that is given by its name. Like for streams, this option supports SWI-Prolog's lazy lists to read in the file.

- `string`: `string("append([], Y, Y).")`
  The given text enclosed in double quote characters is used as the predicate's input.

- `chars`: `chars([a, p, p, e, n, d, '(', '[', ']', ..., ')', '.'])`
  The given list of characters is used as the predicate's input.

Since *library(plammar)* is based on DCGs, all predicates internally work on lists of characters, therefore all input formats are mapped to `chars`. Only the formats `string` and `chars` allow to use the predicates of *library(plammar)* in all instantiation modes, as the conversion from a string to a list of characters and vice versa can be implemented in a logically pure way. Input streams on the other hand cannot be used for writing, which similarly applies for the format `file`.

9.1.3. Command Line Interface

The *library(plammar)* comes with a command line interface to parse given source code and return the list of tokens, parse tree, or AST. It can be executed as follows:

```
swipl -g main cli.pl -- [options] [<filename>]
```
Bash

Called with the flag `--help` instead of the filename lists all available options. The command line interface accepts a filename as the first argument. If called without, the source is read from standard input `stdin`. As usual in command line interfaces, the empty `--` is required to separate the options given to SWI-Prolog (e. g., `-g` to declare `main/0` as SWI-Prolog's initial goal) from those processed by our *library(plammar)* command line interface in `cli.pl`.

It is possible to create a pre-compiled file which increases the tool's performance significantly. The command line interface in `cli.pl` is compiled using SWI-Prolog's `-c` option which creates a saved state [136, Sec. 13.2]:

```
swipl -g main -o cli.exe -c cli.pl && chmod +x cli.exe
```
Bash

The pre-compiled file `cli.exe` is automatically created during the package's installation as a SWI-Prolog add-on, and via the Makefile target `make cli`. The created file's suffix `.exe` is chosen for compatibility with Windows systems. After the pre-compilation step as mentioned before, the created executable can be called via:

```
./cli.exe [options] [<filename>]
```
Bash

9.1.4. Foundations

The *parser* is an important component of compilers and interpreters, as it analyses the source code of a programming language to create some form of internal, hierarchical data representation. It is often preceded by a step for lexical analysis, which first creates a sequence of tokens which the parser works on. This step is thus also called *tokenisation*. With DCGs it is possible to write *scannerless parsers* (also called *lexerless parsers*) that combine these two separate steps into a single grammar. However, the ISO Prolog standard defines Prolog in the similar, traditional way: it first declares that a Prolog program consists of Prolog terms that are a sequence of tokens, and later defines the grammars for tokens (Iso 6.4) and terms (Iso 6.2 & 6.3) separately. Therefore, our implementation of *library(plammar)* is also split into these two phases. Both make use of grammars but work on lists of different types: the lexer handles the program source code as a string – i. e., a list of

characters –, the parser on the other hand works with a list of tokens. This goes well with Prolog's built-in DCG formalism – as introduced in Section 6.2, they allow to define grammars on difference lists of any type.

For performance reasons in the different use cases of parsing on the one hand and serialisation on the other, the tokenisation provided by *library(plammar)* uses both Prolog's DCG formalism, and a finite-state machine. In the following Section 9.2, we focus on the lexer implementation using the syntax description given in the form of EBNF grammars in the ISO Prolog standard. In Section 9.3, a second, alternative approach to implement the lexer as a finite-state machine is presented. The parser component is discussed separately in Section 9.4.

9.2. Tokenisation with the ISO Prolog Standard's EBNF

The syntax of Prolog is described in the ISO Prolog standard using the grammar formalism EBNF. With our considerations from Section 5.6, this grammar can be integrated as an internal DSL in Prolog with only minor modifications:

- In the name of nonterminals, we replace whitespaces by underscore characters. The ISO Prolog standard does not use uppercase letters, therefore the modified nonterminal symbols already constitute valid Prolog atoms and thus callables.
- The very last grammar rule ends with the full stop `.` instead of EBNF's semicolon symbol ";", which can be used as infix operator `;/2`$_{\text{ISO}}$ in Prolog to terminate all but the last EBNF grammar rules.
- Comments are written as usual in Prolog as `/* ... */` instead of EBNF's `(* ... *)`.

This way, the notation for Prolog terms from the EBNF of Iso 6.4 is written using the internal Prolog DSL as follows:

EBNF DSL
```
term = /* 6.4 */
  { token /* 6.4 */ } ;
```

As introduced for EBNF in Section 5.6, `{ token }` denotes the possibly empty repetition of the nonterminal `token//0`. That means that following the ISO Prolog standard, a Prolog term is lexically just a sequence of tokens, whose format is again described by another EBNF grammar rule in Iso 6.4.

Given the Prolog syntax in the form of EBNF rules from the ISO Prolog standard, they can be stated verbatim in a sub-module of *library(plammar)* using the internal

Prolog DSL for EBNF we defined in Section 5.6. An extract of ISO 6.4 that formally describes the syntax of a Prolog variable is given in Appendix B.2.

The overall grammar for all Prolog tokens is implemented in the module *library(plammar/token)*.[37] Following the exact definitions of ISO 6.4, it implements more than 100 nonterminals in about 800 lines of code, using the internal Prolog DSL for EBNF. The *processor character set* of ISO 6.5, which describes Prolog's alphabet (cf. Section 3.1), defines additional 33 nonterminals in about 300 lines of code. The latter include the frequently used nonterminals `small_letter_char//0`, `capital_letter_char//0`, and `decimal_digit_char//0`.

9.2.1. Expanding the Internal DSL into DCGs with Parse Trees

In Appendix B.3, we present a term expansion that translates this internal DSL representation of EBNF into multiple DCGs, one for each nonterminal. For the usage with *library(plammar)*, we expand the DCGs by an additional argument `Opts` for the options list that is passed to *library(plammar)*'s `prolog_parsetree/3`. This is achieved by another term expansion for DCGs that adds the variable `Opts` to all nonterminals, using the predicate `term_args_attached/3`[C.4].

By additionally using our *library(dcg4pt)*, which we introduced in detail in Chapter 8, the definite clause grammars created this way can be further modified to each hold an additional argument for their corresponding parse tree, which is represented by a compound term. Finally, these modified DCG rules are translated into plain old Prolog clauses using the standard term expansion scheme for DCGs, as presented in Section 6.2.4.

Listing 9.1 shows in (a) the ISO Prolog standard's original EBNF that describes a Prolog term, and the following four steps of this source-to-source transformation:

(a) presents the extract of the original EBNF as specified in ISO 6.4 as an internal Prolog DSL,

(b) depicts the equivalent DCG,

(c) shows the DCG extended by the additional argument `Opts` for the options list,

(d) is its modified version with a new last argument that holds the processed parse tree from *library(dcg4pt)*,

(e) shows the created Prolog clause for this modified DCG as generated by SWI-Prolog's standard term expansion for DCGs.

[37] Its sources are located in the project's repository in the Prolog file `prolog/plammar/token.pl`.

Listing 9.1: Formal description of a term's syntax as a sequence of tokens as

(a) internal DSL for EBNF,

```
term = /* 6.4 */
  { token /* 6.4 */ } ;
```

(b) DCG,

```
term --> /* 6.4 */
  *token /* 6.4 */ .
```

(c) DCG with options,

```
term(Opts) --> /* 6.4 */
  *token(Opts) /* 6.4 */ .
```

(d) DCG with options and parse tree,

```
term(Opts, term(PTs)) --> /* 6.4 */
  *(token(Opts), PTs) /* 6.4 */ .
```

(e) Prolog clause with options, parse tree, and difference list.

```
term(Opts, term(PTs), A, Z) :-
  *(token(Opts), PTs, A, Z).
```

Note that the source code annotations cannot be accessed in the term expansion, thus they are actually not part of the created DCGs or Prolog clauses in Listing 9.1 (b)–(e). Nevertheless, in this and the following code examples we transfer the comments as they help the reader to find the ISO Prolog standard's original definition of the referenced nonterminal.

In Section 8.4, we introduced *library(dcg4pt)*'s meta-nonterminal `sequence//2` to denote sequences and optionals of nonterminals. Similarly, *library(plammar)* defines the prefix operators `?/1` (optional element), and `*/1` (possibly empty repetition) for a concise and short notation for sequences in the DCGs of step 2. Since never used by the ISO Prolog standard, there is no dedicated prefix operator for the non-empty sequence. After applying all term expansions, these prefix operators result in the Prolog predicates `?/4` and `*/4`, which in addition to the referenced DCG body also hold arguments for the list of parse trees, and a pair that represents the processed difference list. Their definition is given in Appendix B.9. To support both efficient parsing and serialisation, they necessarily differentiate between predicate calls with a bound parse tree and others with a bound difference list.

Following the ideas of *library(dcg4pt)*, the extended DCG for the nonterminal `term//0` describes a parse tree of the form `term(PTs)`. According to our modified term expansion for sequences of the nonterminal `token//0`, the variable `PTs` will always be a list of parse trees for all token, i. e., each list element is a compound term `token(...)` again. This list of tokens is exactly the result one would expect from a lexer, but also from the predicate `prolog_tokens/3` that ships with *library(plammar)*. Consequently, it is implemented as follows, using `phrase/3`$_{\text{ISO}}$ on the expanded nonterminal `term//2`, with the list of options `Opts` in its first argument, and the processed parse tree `term(Tokens)` in its second:

```prolog
%% prolog_tokens(?Source, ?Tokens, ?Opts)
prolog_tokens(chars(Chars), Tokens, Opts) :-
  phrase(term(Opts, term(Tokens)), Chars, []).
```

Though *library(dcg4pt)* also supports open difference lists (i. e., the very last argument `Z` is a free variable), *library(plammar)* works only on closed difference lists. This is because of its application, which is focussed on Prolog programs. The processed Prolog source code file is finite, and so is the list of characters and the list of tokens. Therefore, the last argument of `phrase/3`$_{\text{ISO}}$ is known to be the empty list `[]`.

9.2.2. Context-Sensitive Requirements

The format of tokens in the programming language Prolog is specified in ISO 6.4. While most of the syntax is described using only the grammar formalism EBNF, the ISO Prolog standard also contains informally specified requirements which cannot be expressed by context-free grammars. For instance, in addition to the grammar rules that describe the language's token lexically, ISO 6.4 also states the following requirement informally:

> *A token shall not be followed by characters such that concatenating the characters of the token with these characters forms a valid token [...].*

This requirement cannot be expressed by a context-free grammar, because it requires an arbitrary look-ahead. Therefore, not all nonterminals are implemented by the internal Prolog DSL. Instead, explicit context-sensitive requirements use one of the formats shown as intermediate steps of the program transformation. The used level depends on whether the requirement relies on (c) just the options list, (d) the parse tree, or (e) the difference list.

For instance, the aforementioned example of tokens, that shall not be followed by characters that again form a valid token, requires access to the subsequent elements of the processed difference list. The look-ahead can be implemented in two ways:

- In some of the intermediate steps (b)–(d) that represent DCGs, the grammar rule can first consume the following characters to ensure they are not part of the currently examined sequence, and than pushed back unchanged to the difference list using DCG's built-in semicontext notation. This follows the idea of the nonterminal `elem//0` which we used as an example in Section 6.2.2.
- Alternatively, the plain old Prolog clause that is created in step (e) allows access to the elements of the processed difference list in its last two arguments.

Listing 9.2: Prolog wrapper `token_/4` for the nonterminal `token//0` to ensure the context-sensitive requirements of ISO 6.4.

```
1 token_(Opts, token(Tree), A, Z) :-                                    PROLOG
2   nonvar(A), !,
3   token(Opts, token(Tree), A, Z),
4   Some_More_Elements = [_|_], % at least one element
5   \+(( token(Opts, _, A, Shorter_Z),
6        append(Some_More_Elements, Shorter_Z, Z) )).
7 token_(Opts, token(Tree), A, Z) :- token(Opts, token(Tree), A, Z).
```

Both alternatives result in similar Prolog predicates. In *library(plammar)*, we opted for the second option, as it additionally allows to differentiate between the cases the Prolog predicates are called with either a bound difference list or bound parse tree for improved performance. In the latter case of a known parse tree, the required look-ahead for the difference list can be skipped, because in our applications we assume a well-formed input that describes a valid Prolog program. Therefore, the list `Tokens` in the parse tree `term(Tokens)` cannot contain two subsequent tokens which together would also form a valid token, making the look-ahead obsolete for this instantiation mode `(+,+,-,-)`.

In Listing 9.2, we present the definition of the Prolog predicate `token_/4`, which serves as a wrapper for `token/4` that is created by the program transformation for EBNF. If the parse tree argument is known and the variable for the difference list `A` is free (l. 7), it simply calls the original predicate. Otherwise, the predicate succeeds only if `token/4` does not succeed for a shorter remainder list `Shorter_Z` (ll. 4–6), as this implies that consuming more elements from the difference list would also result in a valid Prolog token. It uses a variadic number of additional elements, since we cannot rely only on the subsequently following ones. For instance, the character sequences `1` and `1.2` form valid Prolog tokens, as they can be parsed by the nonterminals `integer//0` and `float_number//0`. However, the intermediate character sequence `1.` forms no valid Prolog token.

Since the ISO Prolog standard contains several similar context-sensitive requirements, parsing Prolog is also a prime example for a realistic parser based on DCGs. In our *library(plammar)*, we use the EBNF formalism where feasible. The intermediate DCG representation as well as wrappers for the resulting Prolog predicates are used only if required. For instance, because context-sensitive constraints are informally specified in the ISO Prolog standard, or to enable or disable optional language features which we discuss in Chapter 10.

Listing 9.3: Definition of tokens according to ISO 6.4.

```
1    token = variable | name | integer | float_number                    EBNF DSL
2          | double_quoted_list | open | open_ct | close | open_list
3          | close_list | open_curly | close_curly | ht_sep | comma
4          | back_quoted_string ;
5 variable = [ layout_text_sequence /* 6.4.1 */ ],
6            variable_token /* 6.4.3 */ ;
7     open = layout_text_sequence, open_token /* 6.4.8 */ ;
8  open_ct = open_token /* 6.4.8 */ ;
```

9.2.3. Tokens and Optional Layout Text

In the previous section, we already mentioned two nonterminals for Prolog tokens that represent numbers, `integer//0` and `float_number//0`. Listing 9.3 shows the EBNF from ISO 6.4 that lists all allowed tokens. As an example, Appendix B.2 gives the complete EBNF to parse a variable in the form of the internal Prolog DSL.

In general, Prolog is not whitespace-sensitive, i. e., the program's meaning does not rely on the indentation of the code. The source code can be arbitrarily indented. In the EBNF, this is realised by additionally defining a nonterminal `X_token//0` for each nonterminal `X` on the right-hand side of Listing 9.3, e. g., `integer_token//0` to supplement the nonterminal `integer//0`. Most of these `X_token//0` nonterminals can be preceded by an optional *layout text sequence* (LTS), represented by the nonterminal `layout_text_sequence//0`, as depicted in lines 5–6 of Listing 9.3 for the variable token.

The LTS is also used to resolve ambiguities. Most importantly, it allows to determine whether an atom followed by the left parenthesis `(`, which is represented by the nonterminal `open_token//0`, is the functor of a compound term, or a prefix operator. The first case with no LTS in-between forms the token `open_ct//0`, while the latter with a preceding LTS forms `open//0`. For all other tokens stated in Listing 9.3, the preceding LTS is optional. For instance, `X(1.2.3)` is built by the separate tokens `X` (`variable//0`), `(` (`open_ct//0`), `1.2` (`float_number//0`), `.` (`name//0`), `3` (`integer//0`), and `)` (`close//0`), which is identical for the character sequence `X( 1.2 . 3 )` with additional whitespaces, but slightly changed from `open_ct//0` to `open//0` if there is also a LTS immediately after `X`. Nevertheless, though forming valid Prolog tokens, all these three character sequences can never be part of a valid Prolog term according to the ISO Prolog standard, as we later argue and verify with *library(plammar)* in Section 10.1.5.

Since whitespaces are significant in some of the source code examples in the following sections and Chapter 10, a required whitespace is indicated as `·`. In contrast, normal whitespace characters can always be left without changing the depicted term's meaning. In both cases of `·` and ` `, the LTS can also be built from single line comments `%...` or bracketed comments `/*...*/`. For instance, `p·(1, 2)` can be stated as `p (1, 2)` or `p/* of */(1, 2)`, but not as `p(1, 2)` without the LTS. The first represents a unary compound term `p/1` written in prefix notation, with the pair `(1, 2)` as its single argument, whereas `p(1, 2)` is the compound term of arity 2.

9.2.4. Tokenisation Example: `append/3`

As an example, we consider the following extract of our implementation of the predicate `append/3` from Section 3.4:

```prolog
%% append(?List1, ?List2, ?List1_then_List2)
append([], Y, Y).
append([E|X], Y, [E|Z]) :-
  append(X, Y, Z).
```

Its list of tokens can be created by our *library(plammar)* by calling `?- prolog_tokens(file('append.pl'), Tokens)`. Listing 9.4 shows the elements of the list `Tokens`. Symbols in the first level of indentation depict single tokens, for instance `name//0` (ll. 1–14), `open_ct//0` (l. 15). The inner part of their corresponding parse tree is reasonably indented for better readability. Symbols highlighted in grey boxes depict the parse trees' leaves and thus refer to the processed characters.

The first `name//0` token describes the predicate's name `append` of the program's fact. The preceding single line comment is part of this token's LTS. Because by convention we document a predicate's expected instantiation mode by two leading percent signs `%%`, the second is part of the processed comment text, while the first one is required as the initial *end line comment character* (ISO 6.5.3). Here, the structure of the sub-tree in lines 4–9 strictly follows the definition from ISO 6.4.1: a single line comment is the end line comment character `%`, followed by some commend text in the system's alphabet, and finally ending with the newline character.

Similarly, the other tokens are represented by their corresponding parse tree as created by our *library(dcg4pt)*. The token `open_ct//0` (l. 15) is the only one which does not allow a leading LTS, therefore its inner parse tree is immediately a compound term again instead of a list. The complete first fact of our `append/3` program is built by the tokens of lines 1–25, concluding with the `end_token//0` `.` which

Listing 9.4: Tokens from `append/3`'s source code of Section 3.4 as generated by *library(plammar)*.

Toplevel Output

```
name([
  layout_text_sequence([
    layout_text(
      comment(single_line_comment([
        end_line_comment_char('%'),
        comment_text('% append(?List1, ?List2, ?List1_then_List2)', [
          char(solo_char(end_line_comment_char('%'))),
          char(layout_char(space_char(' '))), ... ]),
        new_line_char('\n') ]))) ]),
  name_token(append,
    letter_digit_token([
      small_letter_char(a),
      alphanumeric_char(
        alpha_char(letter_char(small_letter_char(p)))), ... ])) ]),
open_ct(open_token(open_char('('))),
open_list([ open_list_token(open_list_char('[')) ]),
close_list([ close_list_token(close_list_char(']')) ]),
comma([ comma_token(comma_char(',')) ]),
variable([
  layout_text_sequence([layout_text(layout_char(space_char(' ')))]),
  variable_token('Y', named_variable([capital_letter_char('Y')]))]),
comma([ comma_token(comma_char(',')) ]),
variable([ ... as in lines 19-21 ...  Y ]),
close([ close_token(close_char(')')) ]),
end([ end_token(end_char('.')) ]),
name([ % beginning of second clause
  layout_text_sequence([
    layout_text(layout_char(new_line_char('\n')))]),
  name_token(append,
    letter_digit_token([ small_letter_char(a), ... ])) ]),
...
```

typically terminates a Prolog clause. The second clause begins immediately after. Consequently, the following newline character is part of its LTS.

For practical applications, it has proven to be useful to further extend the parse tree generated by *library(dcg4pt)* by an additional argument for all comments (nonterminals `single_line_comment//0` and `bracketed_comment//0`), which stores the processed comment text. Without, the parse tree provides direct access only to the hierarchical, verbose term structure that follows the definitions of the ISO Prolog standard. Given this compound term, the original text can be created again using *library(plammar)*'s DCGs the other way round, as for our example:

```
?- PTs = [char(...), char(...), ...],                    TOPLEVEL
   comment_text(_Opts, comment_text(PTs), Text, []).
Text = "% append(?List1, ?List2, ?List1_then_List2)" .
```

This typical requirement of real-world applications creates an unnecessary overhead, which we avoid in *library(plammar)*. The processed character sequence is instead stored directly in the parse tree for comments. Similarly, the processed character sequences are stored as an additional argument in the parse tree for the following nonterminals:

- `variable_token//0`,
- `name_token//0` (includes character sequences enclosed in `'...'`),
- `integer_token//0`,
- `float_number_token//0`,
- `double_quoted_list_token//0` (strings enclosed in `"..."`),
- `back_quoted_string_token//0` (character sequences enclosed in `` `...` ``),

The classical term expansion scheme for DCGs does not allow to accumulate the processed elements of the difference list. We therefore define in *library(plammar)* a suffix operator `wrap/1` to denote nonterminals that should produce the additional argument. Then, the EBNF for the nonterminal `comment_text//0` becomes as follows:

```
comment_text wrap = [ char /* 6.5 */ ] ;                 EBNF DSL
```

As a result, in Listing 9.4 the compound terms for the nonterminals `comment_text//0`, `name_token//0`, and `variable_token//0` hold the text their parse tree represent in their first argument, e. g., in lines 6 and 10.

9.3. Tokenisation with a Finite-State Machine

Though the formal notation in the form of EBNF as an internal DSL allows the syntax of the programming language Prolog to be described declaratively and yet directly executable as a Prolog program, the grammar can be improved to parse real-world applications more efficiently. Our *library(plammar)* therefore alternatively implements the lexer as a finite-state machine for the use case of serialisation. For the definition of the processed character set, the finite-state machine still relies on the EBNF given in the form of an internal DSL. This approach is also used for all calls of *library(plammar)* predicates with a bound parse tree.

9.3.1. Addressed Problems

In an implementation of the lexer that is based only on EBNF as described in Section 9.2, the following three major enhancements are feasible, improving the lexer's performance and its memory footprint.

Common Prefixes. Without modifications, the grammar rules given in the ISO Prolog standard contain some nonterminals with identical prefixes on the rules' right-hand sides. Most notably, this is related to the handling of the LTS, which is allowed to precede almost all token, as shown in Listing 9.3. Following Prolog's SLD resolution mechanism, this makes the parsing of the LTS redundant and is repeated again for every alternative token type. For instance, given the term `·1.2`, the LTS depicted by `·` is first parsed as part of the nonterminal `variable//0`, which fails for `variable_token//0`. For the next alternative in the grammar rule's right-hand side, the LTS is parsed all over again, and so on, until finally the parsing succeeds for the nonterminals `float_number//0` and `float_number_token//0`.

Besides the repeated parsing of the LTS, this example underlines that there are also other sources of repeated computations for identical prefixes on the EBNF rules' right-hand sides. The term `·1.2` first succeeds for the nonterminals `integer//0` and `integer_token//0`, which is backtracked only after checking the context-sensitive requirements we discussed in Section 9.2.2: a Prolog token is valid only if combined with some subsequent characters does not also lead to a valid token, which in our example is given for `1.2` being a valid `float_number//0` token. For this alternative, the leading numbers in front of the decimal symbol `.`, are finally parsed again.

Dependency on `append/3`. The handling of optionals and sequences of nonterminals by our *library(dcg4pt)* heavily relies on the predicate `append/3`[3.4] to

combine the parse trees created by the elements in a grammar rule's right-hand side into a single compound term. This is mainly due to *library(dcg4pt)*'s and *library(plammar)*'s intended use for both parsing and serialisation, which requires logically pure predicates. As described in Section 8.4.3, we therefore avoid logically impure predicates like `flatten/2`[C.12]. The same applies for the concept of difference lists, which can conversely not be used to split a list (cf. Section 3.4). But since `append/3`[3.4] is linear to the length of the first argument, the parsing performance significantly decreases for longer sequences. While in real-world Prolog programs this has only little effect with respect to the parsing of, e. g., variable names, atoms, and numeric values, we encountered in our empirical community evaluation [82] several well documented Prolog programs with large source code annotations. Relying on `append/3`[3.4] here to create the parse tree for the LTS results in a huge parsing overhead, which further increases because of the repeated processing of the LTS as described before.

Non-Optimised Recursions. Not only because of our dependency on `append/3` in the used grammars, the Prolog clauses created from the definite clause grammars by *library(dcg4pt)* and Prolog's standard term expansion scheme are not available for SWI-Prolog's tail call optimisations (sometimes also referred to as "tail recursion optimisations" in the context of Prolog). Though this does not affect the lexer's performance, it results in a larger memory consumption.

Given that, it is clear that though our lexer implementation of Section 9.2, which solely relies on the EBNF grammar given in the ISO Prolog standard, provides a correct result and produces the corresponding parse tree as required, it does not constitute an efficient lexer in Prolog. The first aspect of common prefixes alone can be improved by manually re-organising the grammar rules provided by the ISO Prolog standard, which is called *factoring out*: the common prefix is put in a separate nonterminal which is parsed first, followed by the various alternatives.

However, with *library(plammar)* we opted for a dedicated finite-state machine instead, using the common prefixes as intermediate steps when parsing. In addition, it represents the tokens as a difference list, resulting in an efficient lexer. Because on its reliance on difference lists, this finite-state machine is used only for the predicate's application of parsing, i. e., in instantiation modes where the difference list is bound. Unlike the more general use cases of *library(dcg4pt)*, no splitting of lists is required in case of tokenisation.

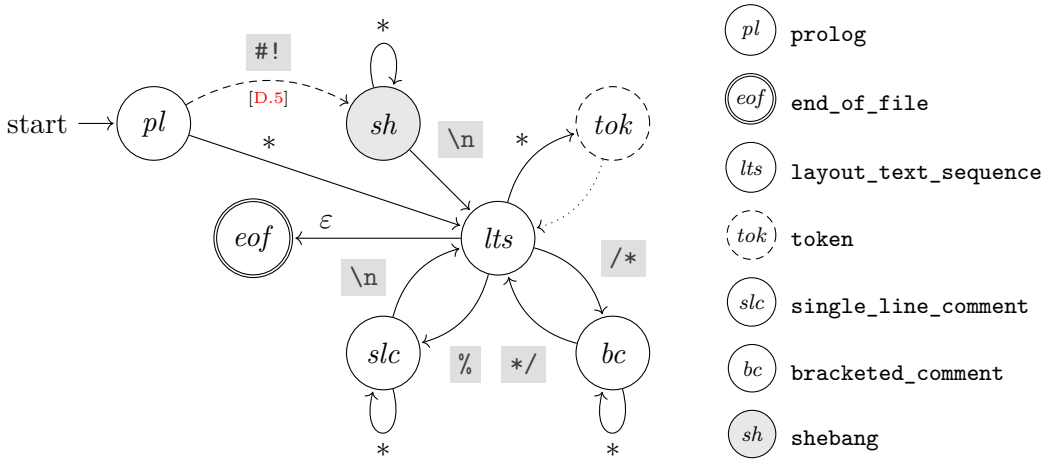

Figure 9.1: Finite-state machine for handling the LTS that precede Prolog tokens.

9.3.2. Handling of Layout Text

The three aspects for improvement we summarised in Section 9.3.1 mainly address the problems that arise from combining read characters into tokens and the LTS, and are therefore only related to ISO 6.4. For the description of Prolog's alphabet with its processed character set on the other hand, as it is defined in ISO 6.5, our finite-state machine still uses the DCGs produced from the EBNF rules by the program transformations we introduced in Section 9.2. Their grammar rules do not suffer from common prefixes or sequences which require the usage of `append/3`[3.4]. For instance, the nonterminal `decimal_digit_char//0` (ISO 6.5.2) just lists all allowed possibilities, i. e., `0` – `9`. Its definition does not profit from an alternative implementation. We therefore replace only the definition of the tokens according to ISO 6.4, as previously depicted in the internal DSL in Listing 9.3, by a finite-state machine.

Figure 9.1 gives an overview of the processing of tokens and their preceding LTS. The names of the states are abbreviated in the graph. Each edge depicts a state transition that is applied if the denoted character sequence (highlighted in grey) is encountered in the processed difference list; otherwise, the path denoted by the wildcard symbol * is used.

The initial state for a Prolog program is `prolog` (*pl*). It allows to distinguish the beginning of a Prolog program from the rest. For instance, it is used by the Prolog language extension `allow_shebang`[D.5], which we present in Section 10.2.2. If the program does not start with the character sequence `#!`, we continue with the state `layout_text_sequence` (*lts*). From here, the finite-state machine processes the given difference list by moving between the states for the `single_line_comment` (*slc*), the `bracketed_comment` (*bc*), and the `token` (*tok*). The latter relates to the `X_token//0`

nonterminals of the EBNF, i. e., Prolog's smallest lexical element. This way, instead of parsing the preceding LTS separately for each token, all layout text and source code annotations are processed first. Only after the handled character sequence is empty ε, the final state `end_of_file` (*eof*) is reached.

9.3.3. State Transition Rules for Tokens

The shape of the processed Prolog tokens is manifold. Consequently, there are many different paths from the state `token` (*tok*) back to `layout_text_sequence` (*lts*), which is therefore depicted in Figure 9.1 by a dotted edge. These paths are first determined by the class of the following characters – for instance, a lowercase letter `a`–`z` introduces a `letter_digit_token//0`, which constitutes a *name* in Prolog (ISO 6.4.2). A decimal digit `1`–`9` on the other hand indicates a number, though it is not yet determined if it is an integer (nonterminal `integer//0`), or fractional (`float_number//0`). Even more different tokens can be started by the decimal digit `0`. Because of Prolog's short notations for binary numbers, octal numbers, hexadecimal numbers, and character codes, at least two more characters are required to distinguish the valid tokens. For instance, `0xf` denotes the single token for the hexadecimal number 15, while `0xg` constitutes the two tokens `0` (`integer//0`) and `xg` (`name//0`).

Table 9.1 lists all character classes with their created token and the following state. The middle column indicates the functor of the created parse tree. For some prefixes, its outer term is the same – e. g., `name_token(...)` if the difference list continues by a lowercase letter, as well as if it is the semicolon `;`. In this case, Table 9.1 further determines the parse tree's second level, i. e., the parsed character `;` produces a Prolog term of the form `name_token(semicolon_token(...))`.

Besides the state `layout_text_sequence` (*lts*), we introduce additional states used in the finite-state machine of the lexer. Their name indicates their use case. For instance, the state `seq_alphanumeric_char` constitutes a sequence of characters `A`–`Z`, `a`–`z`, `0`–`9`, and the underscore `_`. Unlike the corresponding DCG nonterminals produced by *library(dcg4pt)*, the processing of these repetitions relies on difference lists instead of the idioms introduced in Section 8.4.

The definitions of the most common sequences is given in Table 9.2. For a given state and prefix in the processed difference list, it lists the following state, and additional conditions where necessary. For instance, the state `seq_hexadecimal_digit_char` can also contain underscores and spaces in case the language extensions `allow_digit_groups_with_{space,underscore}`[D.8,D.9] are enabled. If none of the provided prefixes is used, the lexer continues in the state `layout_text_sequence` (*lts*).

Table 9.1: Transition rules in the finite-state machine for the state `token`.

Prefix	Token	Following State
`_`	`variable/anonymous_variable` *for trailing* LTS	`layout_text_sequence`
`_`	`variable/named_variable`	`seq_alphanumeric_char`
`A` – `Z`	`name/letter_digit` *for* ***var_prefix***[D.7]	`seq_alphanumeric_char`
`A` – `Z`	`variable/capital_variable`	`seq_alphanumeric_char`
`a` – `z`	`name/letter_digit`	`seq_alphanumeric_char`
`'`	`name/quoted`	`seq_single_quoted_item`
`"`	`name/double_quoted_list`	`seq_double_quoted_item`
`` ` ``	`name/back_quoted_text` *for* ***back_quoted_text***[D.6]	`seq_back_quoted_item`
`;`	`name/semicolon`	`layout_text_sequence`
`!`	`name/cut`	`layout_text_sequence`
`,`	`comma`	`layout_text_sequence`
`|`	`ht_sep`	`layout_text_sequence`
`[`	`open_list`	`layout_text_sequence`
`]`	`close_list`	`layout_text_sequence`
`{`	`open_curly`	`layout_text_sequence`
`}`	`close_curly`	`layout_text_sequence`
`(`	`open` *for preceding* LTS `open_ct` *for empty* LTS	`layout_text_sequence` `layout_text_sequence`
`)`	`close`	`layout_text_sequence`
`.`	`end`	`layout_text_sequence`
`.`	`name/graphic` *if followed by one of the following graphic tokens*	`seq_graphic_token_char`
`#`, `$`, `&`, `*`, `+`, `-`, `/`, `:`, `<`, `=`, `>`, `?`, `@`, `^`, `~`	`name/graphic`	`seq_graphic_token_char`
`0'`	`integer/character_code`	`single_quoted_char`
`0b`	`integer/binary_constant`	`seq_binary_digit_char`
`0o`	`integer/octal_constant`	`seq_octal_digit_char`
`0x`	`integer/hexadecimal_constant`	`seq_hexadecimal_digit_char`
`0` – `9`	`integer` *or* `float_number`	`number`, *cf. Figure 9.2*

Table 9.2: State transition rules for some important character groups. If no prefix matches or the given condition is not satisfied, the automata continues in the state `layout_text_sequence`.

State	Prefix	Conditions	Following State
`single_quoted_char`	`'`	`allow_single_quote_char_in _character_code_constant`[D.12]	`layout_text_sequence`
`seq_alphanumeric_char`	`_`, `A`–`Z`, `a`–`z`, `0`–`9`		`seq_alphanumeric_char`
`seq_binary_digit_char`	`0`, `1`		`seq_binary_digit_char`
	`_0`, `_1`	`allow_digit_groups_with_underscore`[D.9]	`seq_binary_digit_char`
	` 0`, ` 1`	`allow_digit_groups_with_space`[D.8]	`seq_binary_digit_char`
`seq_octal_digit_char`	`0`–`7`		`seq_octal_digit_char`
	`_0`–`_7`	`allow_digit_groups_with_underscore`[D.9]	`seq_octal_digit_char`
	` 0`–` 7`	`allow_digit_groups_with_space`[D.8]	`seq_octal_digit_char`
`seq_decimal_digit_char`	`0`–`9`		`seq_decimal_digit_char`
	`_0`–`_9`	`allow_digit_groups_with_underscore`[D.9]	`seq_decimal_digit_char`
	` 0`–` 9`	`allow_digit_groups_with_space`[D.8]	`seq_decimal_digit_char`
`seq_hexadecimal_digit_char`	`0`–`9`, `A`–`F`, `a`–`f`		`seq_hexadecimal_digit_char`
	`_0`–`_9`, `_A`–`_F`, `_a`–`_f`	`allow_digit_groups_with_underscore`[D.9]	`seq_hexadecimal_digit_char`
`seq_graphic_token_char`	`#`, `$`, `&`, `*`, `+`, `-`, `/`, `:`, `<`, `=`, `>`, `?`, `@`, `^`, `~`		`seq_graphic_token_char`

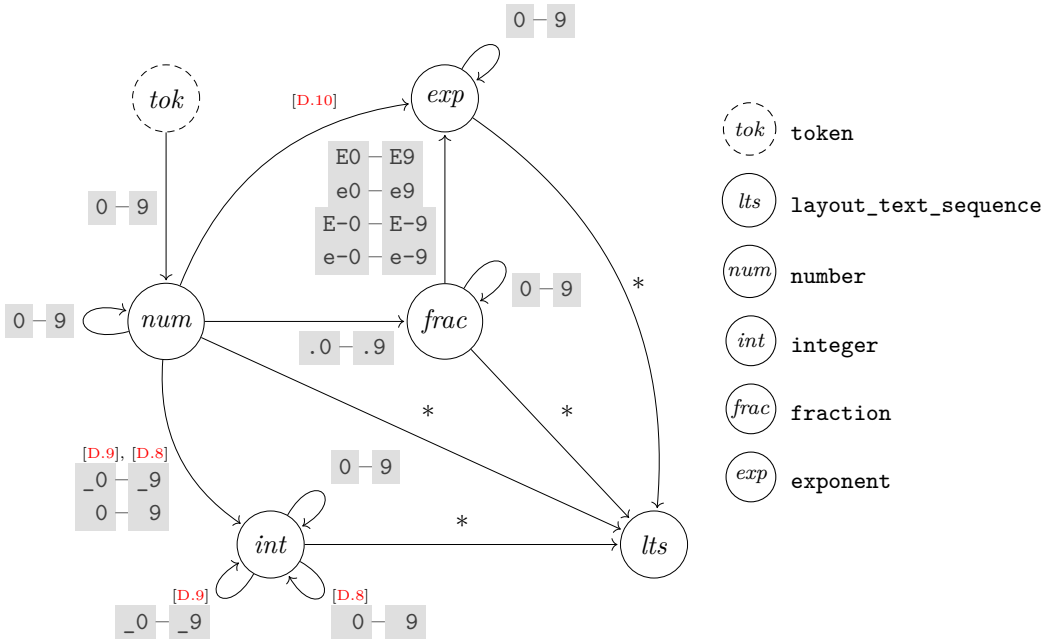

Figure 9.2: Extract of the finite-state machine for parsing numbers.

9.3.4. Parsing of Numbers

As mentioned before, parsing numbers in Prolog just as described in ISO 6.4.4 & 6.4.5 possibly creates unnecessary backtracking, because of the common prefix on the right-hand sides of the nonterminals `integer//0` and `float_number//0`. Figure 9.2 shows an extract of the finite-state machine that is used for parsing numbers, beginning with the state `token` (*tok*), and ending in the state `layout_text_sequence` (*lts*) again.

Figure 9.2 describes the different paths to constitute a valid number token in Prolog:

– An integer is built from only decimal digits. The parse tree is of the form `integer_token(...)`.
(*tok*) – (*num*) – (*lts*)

– A fractional is built from decimal digits, followed by `.`, and again decimal digits. The parse tree is of the form `float_number_token(...)`.
(*tok*) – (*num*) – (frac) – (*lts*)

– Additionally, the `float_number_token` can be extended by an exponent.
(*tok*) – (*num*) – (frac) – (*exp*) – (*lts*)

– Following ISO 6.4.5, the exponent is allowed only after the fractional part. This requirement is relaxed by some Prolog systems, so that `1e3` becomes a valid `float_number_token`. In *library(plammar)*, it can be enabled by the option `allow_integer_exponential_notation`[D.10].
(*tok*) – (*num*) – (*exp*) – (*lts*)

– SWI-Prolog supports splitting long integers into digit groups [136, Sec. 2.16.1.5], both by single spaces or underscores between decimal digits. *library(plammar)* provides similar language extensions, which we present in Section 10.2.2. It is not allowed to mix this notation for long integers with floating-point numbers, therefore we introduce a separate state `integer` (*int*).
(*tok*) – (*num*) – (*int*) – (*lts*)

9.3.5. Implementation in Prolog

The finite-state machine is implemented in *library(plammar)* in the form of the predicate `tokens/5`:

```
%% tokens(?Opts, +State, -Tokens, +A, ?LTS)                    PROLOG
```

Since the processed difference list `A-Z` is known to be closed, we omit the empty list `Z` in the list of arguments for `token/5`. Given the bound list of characters in `A`, it binds the variable `Tokens` to the list of tokens. The list of options `Opts` can be given or its settings inferred. The variable `State` determines the current state and is initially bound to the atom `prolog`. The names of all other states follow the previously introduced finite-state machines, i. e., (*lts*) is encoded as `lts`.

The very last argument `LTS` of `tokens/5` serves multiple purposes. In the state `lts`, it is a difference list that gradually collects the individual parse trees for the processed layout text, starting from the empty difference list `L-L` (an open list `L` with the remainder `L`). It uses the nonterminal `layout_text//0` as defined using the internal Prolog DSL for EBNF (ISO 6.4.1). As a result, for the input `/**/`, the following difference list for the LTS is created:

```
[ layout_text(layout_char(space_char(' '))),                    PROLOG
  layout_text(comment(bracketed_comment(...)),
  layout_text(layout_char(space_char(' '))) |R]-R
```

The closed list (i. e., with `R=[]`) is passed to all following states in the last argument `LTS` of `tokens/5`, until the state `lts` is reached again at some point, which starts with a fresh difference list again. Passing LTS to the other states is required

Listing 9.5: Backtracked solutions for the tokenisation of the input `1_0`.

```
?- _Opts = [allow_digit_groups_with_underscore(YesNo)],          TOPLEVEL
   prolog_tokens(string("1_0"), Tokens, _Opts).
① YesNo = true, Tokens = [
     integer([integer_token('1_0', integer_constant([
       decimal_digit_char(1),
       underscore_char('_'),
       decimal_digit_char(0) ])) ]) ] ;
② YesNo = false, Tokens = [
     integer([integer_token(1, integer_constant([
       decimal_digit_char(1)]))]),
     variable([variable_token('_0', named_variable([
       variable_indicator_char(underscore_char('_')),
       alphanumeric_char(decimal_digit_char(0)) ])) ]) ].
```

to differentiate between the tokens `open//0` and `open_ct//0`, as well as because the LTS is part of the created parse tree of the following token – as depicted in lines 2–9 of Listing 9.4 in the exemplary parse tree for the `append/3`[3.4] Prolog program.

Each clause of `tokens/5` in our *library(plammar)* constitutes another state. All in all, the Prolog implementation of the finite-state machine relies on 36 different states. Their state transitions are defined in more than 500 lines of Prolog code. Following our argumentation for using a dedicated lexer implementation based on a finite-state machine at the beginning of this Section 9.3, it is known that the predicate `tokens/5` is called with only the aforementioned instantiation mode, i. e., with a bound character list `A`. However, our implementation still avoids logically impure predicates like the cut $\texttt{!/0}_{\texttt{ISO}}$. Consequently, the tokenisation can still be backtracked. For instance, for the input `1_0`, *library(plammar)* returns two possible ways of tokenisation for an unknown value for the option `allow_digit_groups_with_underscore`[D.9], as shown in Listing 9.5. In the first computed answer substitution ①, it is parsed as a single integer that constitutes the number 10, using the notation for long integers with digit groups separated by the underscore `_`. The second answer in ② is compliant to the ISO Prolog standard instead. It parses the input as the integer 1, immediately followed by the named variable `_0`.

9.4. Term Parsing

Just like tokens are described in the ISO Prolog standard using EBNF grammars that describe sequences of characters, ISO 6.3 formally defines what sequences of tokens constitute valid Prolog terms. They are the building blocks of clauses and

Listing 9.6: Definition of Prolog clauses and program text according to ISO 6.2.

EBNF DSL

```
  prolog_text = p_text ;
       p_text = directive_term, p_text ;
              | clause_term, p_text ;
  clause_term = term(_), end ;
directive_term = term(_), end ;
```

directives, and after all complete Prolog programs. Valid Prolog source code is simply called *Prolog text* in the ISO Prolog standard. It is defined by the nonterminal `prolog_text//0` in ISO 6.2, together with the formal grammar for clauses and directives.

We can adapt the ideas from the tokenisation component for the Prolog term and program parsing. It is as described before – only that tokens created by `prolog_tokens/`$\{2,3\}$ are processed by the DCGs rather than characters. The Prolog clauses are again based on the EBNF grammar given in the ISO Prolog standard. They are specified in our internal Prolog DSL, which is then transformed into DCGs and finally plain old Prolog clauses, performing the same program transformations we presented for the tokenisation component in Section 9.2.

The nonterminal `prolog_text//0` serves as an entry point for the predicates `prolog_parsetree/`$\{2,3\}$ that are provided by *library(plammar)* to parse a given string, stream, or Prolog source code file into its corresponding parse tree, or use it vice versa for the serialisation of a given parse tree into the corresponding source code. Listing 9.6 shows the nonterminal's definition from ISO 6.2 in the form of the internal Prolog DSL for EBNF. Everything that can be parsed as a valid term also constitutes a valid Prolog clause together with the `end//0` token. Directives are defined similar, only that the principal functor of this term is required to be `:-/1`$_{\text{ISO}}$. From a syntactical point of view, there is no difference between clauses and directives.

Parsing Prolog source code requires to annotate all terms by their priority. Therefore, the referred nonterminal `term//1` is of arity 1, with a free variable as its argument. In the definitions of the nonterminals `clause_term//0` and `directive_term//0` it is a free variable, since the actual priority of this term does not matter, and can be anything from 0 to 1200. A general overview of the meaning of term priorities and operator precedences was already given in Section 5.1.1.

Every Prolog term is either an atomic term (ISO 6.3.1), a variable (ISO 6.3.2), a curly bracketed term `{...}` (ISO 6.3.6), a list (ISO 6.3.5), a term in *double quoted list notation* enclosed by `"..."` (ISO 6.3.7), or a compound term. Compound terms

Listing 9.7: Definition of numbers as terms according to ISO 6.3.

EBNF DSL

```
1 /* 6.3.1.1 */ term(0) = [ integer(_) ]
2                       | [ float_number(_) ] ;
3 /* 6.3.1.2 */ term(0) = negative_sign_name,
4                         [ integer(_) ]
5                       | negative_sign_name,
6                         [ float_number(_) ] ;
7    negative_sign_name = [ name([name_token('-', _)]) ] ;
8                       | [ name([_, name_token('-', _)]) ] ;
```

written in functional notation (ISO 6.3.3) are of priority 0. Given in operator notation (ISO 6.3.4), their priority is identical to the precedence of the compound term's principal functor, i. e., ranging from 1 to 1200.

All other Prolog terms – most importantly variables, numbers, and strings written in double quoted list notation – are always of priority 0. The same applies for all atoms that are not defined as operators using `op/3`$_{\texttt{ISO}}$; otherwise, the standalone operator is of the highest priority 1201. This constraint ensures that an operator is not the immediate operand of another operator. As a result, the character sequence `\+ / 1` is not a valid Prolog term, though the slash `//2`$_{\texttt{ISO}}$ is an infix operator, and `\+` and `1` on their own are valid Prolog terms. However, it is always possible to explicitly put a term into parentheses, which resets its priority to 0, i. e., `(\+) / 1` becomes a valid Prolog term again. Another alternative is to allow operators as operands, as proposed by our language extension `allow_operator_as_operand`[D.19].

The grammar to process the list of tokens generated in the lexical analysis is specified in *library(plammar/parser)*.[38] Following the exact definitions of ISO 6.2 & 6.3, it implements 70 nonterminals in about 650 lines of code, using the internal Prolog DSL for EBNF. An extract of this grammar from ISO 6.3.1 is given in Listing 9.7, which describes the parsing of numbers. Similar to the notation of DCGs, elements given in square brackets `[...]` represent terminals, i. e., the parse trees like `integer(...)` and `float_number(...)` are elements in the list of tokens generated by the lexer. Following this notation, a term is built from the `integer//0` token (l. 1), or the `float_number//0` token (l. 2), or alternatively each preceded by the nonterminal `negative_sign_name//0` (ll. 3–6) in case of negative numbers. This also underlines why the finite-state machine for tokenising numbers, which we presented in detail in Section 9.3.4, does not include a path for negative numbers – they are constituted as a single term only in the parsing step.

[38] The sources of *library(plammar/parser)* are located in the project's repository in the Prolog file `prolog/plammar/parser.pl`.

The two cases in the definition of the nonterminal `negative_sign_name//0` (ll. 7–8) are required because the name token `-` can also be preceded by a LTS, whose parse tree would be given as the first list element. Following our definition of *library(dcg4pt)* from Chapter 8, terminals are put verbatim into the processed parse tree. In the previous sections, this effected only single characters. With the elements like `integer(_)` in Listing 9.7, the complete nested compound term is part of the parse tree created for `term//0`.

Recall the example from Section 9.2.4. Listing 9.8 shows an extract of the parse tree for our implementation of the predicate `append/3` from Section 3.4. It is created by the goal `?- prolog_parsetree(file('append.pl'), PT)`. Lines 2–23 depict the first fact from the `append/3` program. Following the definition of ISO 6.3.3, the compound term is the sequence of an atom, the opening parenthesis `open_ct`, a list of arguments (denoted by the nested parse tree with the functor name `arg_list`), and the closing parenthesis.

9.5. Towards an Abstract Syntax Tree

The generic structure of the parse tree is based on the grammar's nonterminals, which allows to easily follow the application of the EBNF rules in the ISO Prolog standard when parsing Prolog programs. However, the produced concrete syntax tree (CST) is verbose. In many applications, only the abstract syntax tree (AST) is of interest. It contains only the parts of the parse tree that are required to compile or interpret the depicted program. Consequently, all layout information – i. e., information about indentations, line breaks, and source code annotations – is removed.

For this purpose, our *library(plammar)* provides the predicates `prolog_ast/`$\{2,3\}$. They parse a Prolog source code snippet and return its corresponding AST. Besides the removal of all LTS, it transforms common parse tree structures into more compact nested Prolog terms. This way, the CST that represents a compound term – the sequence of an `atom`, the `open_ct`, a nested structure for its arguments, and finally the `close` token, as for instance given in lines 3–22 of Listing 9.8 – is translated into the shorter AST term `compound(`*`Functor`*`, `*`ListOfArguments`*`)`. Taking up on our running example, Listing 9.9 shows the complete AST for the `append/3` Prolog program, as returned by the goal `?- prolog_ast(file('append.pl'), AST)`.

As for the predicates `prolog_tokens/`$\{2,3\}$ and `prolog_parsetree/`$\{2,3\}$, their AST-related counterpart `prolog_ast/`$\{2,3\}$ can also be used the other way round for the serialisation of a given tree. However, this transformation is underdetermined, as new line breaks and possible indentations have to be arbitrarily added. In

Listing 9.8: Parse tree from `append/3`'s source code of Section 3.4 as generated by *library(plammar)*. The result of the tokenisation is given in Listing 9.4.

Toplevel Output

```
prolog([
  clause_term([
    term([
      atom(name([
        layout_text_sequence(...),
        name_token(append, letter_digit_token(...)) ])),
      open_ct(open_token(open_char('('))),
      arg_list([
        arg(term(atom([
          open_list([ open_list_token(...) ]),
          close_list([ close_list_token(...) ]) ]))),
        comma([ comma_token(comma_char(',')) ]),
        arg_list([
          arg(term(variable([
            layout_text_sequence(...),
            variable_token('Y', ...) ]))),
          comma([ comma_token(comma_char(',')) ]),
          arg_list(
            arg(term(variable([
              layout_text_sequence(...),
              variable_token('Y', ...) ])))) ]) ]) ]),
      close([ close_token(close_char(')')) ]) ]),
    end([ end_token(end_char('.')) ]) ]),
  clause_term([ % beginning of second clause
    term(xfx, [...]) ])
  ...
```

Listing 9.9: Abstract syntax tree from `append/3`'s source code of Section 3.4 as generated by *library(plammar)*. Its parse tree is given in Listing 9.8.

Toplevel Output

```
prolog([
  fact(compound(
    atom(append),
    [ list([], eol), variable('Y'), variable('Y') ])),
  rule( % beginning of second clause
    compound( % head
      atom(append),
      [ list([ variable('E') ], variable('X')),
        variable('Y'),
        list([ variable('E') ], variable('Z'))
      ]),
    [ compound( % elements in the rule body
        atom(append),
        [ variable('X'), variable('Y'), variable('Z') ]) ]) ])
```

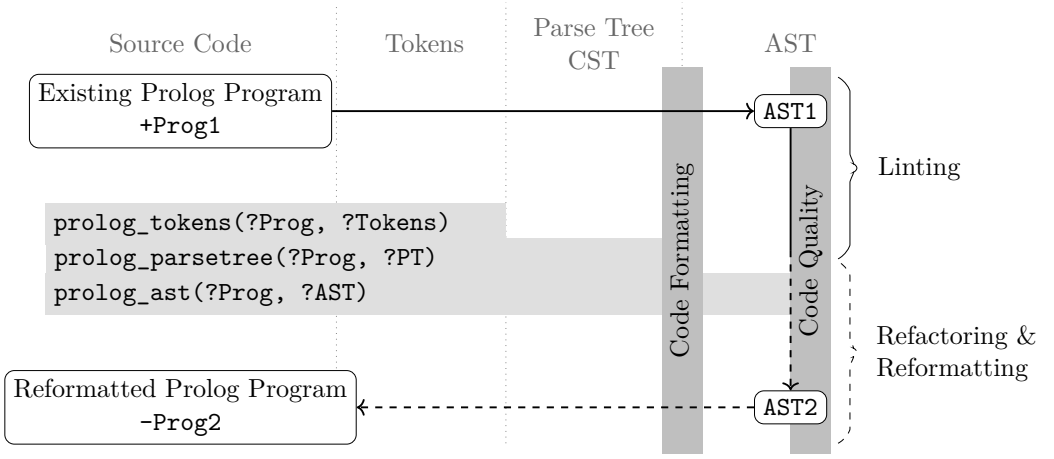

Figure 9.3: Overview of the architecture of a tool that reformats Prolog source code and checks the adherence to coding conventions using *library(plammar)*.

the `Opts` list in the third argument of `prolog_tokens/3`, it is possible to provide an option `style_option(`*`ListOfStyles`*`)`. There, many different coding formatting conventions can be specified. For instance, if the option `newline_after_rule_op` is set to `true`, a line break is added after the clause's principal functor `:-/2`$_{\texttt{ISO}}$. For a given Prolog source code snippet, these coding conventions can be similarly inferred by providing free variables in the `style_option` list.

In our work [82], this mechanism has been used in an empirical community evaluation to automatically check the adherence to Prolog coding guidelines in the form of a code linter. The well-known coding conventions of Covington et al. [25] include constraints regarding the correct code formatting on the one hand, and those with respect to the code quality on the other. With *library(plammar)*, the first class of checks is performed while transforming the CST into the corresponding AST, whereas the second just uses the created AST, as it provides all the necessary information to, e. g., check naming conventions for predicates and variables. As a result, possible code smells are detected as a by-product at each transformation step of *library(plammar)*.

The transformations from a Prolog source code snippet to its corresponding AST, and its serialisation back to a string also brings the opportunity to implement a tool that automatically refactors and reformats Prolog source code. An overview of the architecture of such a tool is shown in Figure 9.3. With two consecutive calls `?- prolog_ast(+Prog1, -AST)` and `?- prolog_ast(-Prog2, +AST, +Opts)` that are connected by the chaining variable `AST`, we can for instance automatically add whitespace characters after every comma in the `arg_list` of a compound term. The created code listing can be adjusted according to our needs by changing the settings in the `style_option` list in `Opts`.

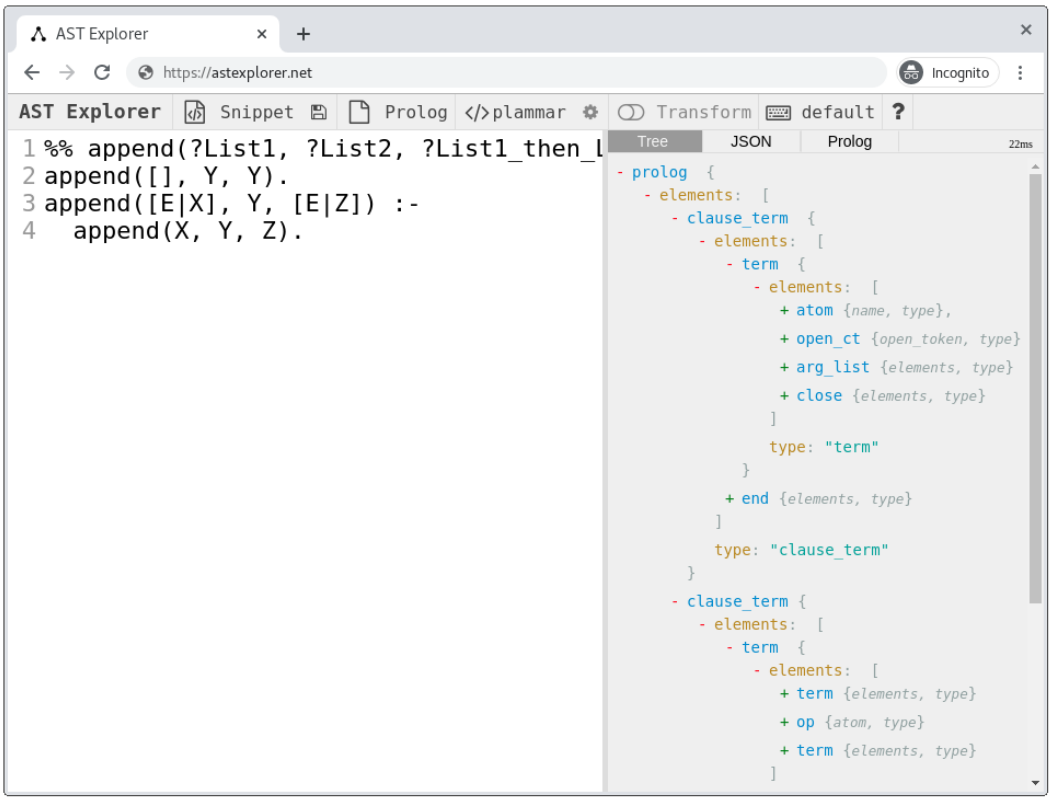

Figure 9.4: Screenshot of an adapted version of the web-based *AST Explorer* for *library(plammar)*.

The transformation rules from a CST to AST and vice versa are implemented in the sub-module *library(plammar/pt_ast)*.[39] Together with the Prolog clauses that check and produce the source code's indentation, this module sums up to more than 1200 lines of Prolog code.

As part of our contribution, we also integrated the Prolog parser *library(plammar)* into the *AST Explorer*, https://astexplorer.net/, an open-source web application that provides parsers for several programming languages, including PHP, JavaScript, and SQL. Figure 9.4 presents the graphical representation of the generated parse tree for the Prolog program of Section 3.4 that defines the `append/3` predicate. Based on SWI-Prolog's *Pengines* [70] infrastructure (cf. Sections 4.3.6 and 7.4), *library(plammar)* ships with a simple web server that listens for HTTP POST requests that transfer Prolog source code snippets in their message body. As its response, the corresponding CST or AST is returned as a JSON document of the format expected by the *AST Explorer*.

[39] The sources of *library(plammar/pt_ast)* are located in the project's repository in the Prolog file `prolog/plammar/pt_ast.pl`. The rules to check the general format of the Prolog source code – e. g., if each Prolog clause consists of a maximal number of subgoals – are split into the separate file `format_check.pl`; all rules regarding the indentation are located in the file `format_space.pl`.

In addition to the integration into the *AST Explorer*, the *Pengines* endpoint can also be used directly. For instance, the following *cURL*[40] command returns the CST representation as a Prolog term for the input program `and(1,X,X)`:

```
curl -H "Accept: text/x-prolog" -H "Content-Type: plain/text" \
  -X POST -d 'and(1,X,X).' http://localhost:8081/
```
BASH

By changing the HTTP `Accept`-header from `text/x-prolog` to `application/json`, the JSON representation needed for the *AST Explorer* is returned instead.

[40] cURL is a command line tool for sending and receiving data including files using URL syntax, typically used with the HTTP protocol. As of today, it is pre-installed in most UNIX-based operating systems.

10

Internal DSL Integration with Operator Inference and Language Extensions

We employ an empirical approach, using optical character recognition software, which finds that merely 93 % of paint splatters parse as valid Perl.

— Colin McMillen[41]

The integration of a DSL internally in Prolog is typically much easier in the course of the language's first definition, as the DSL can then be tailored to the few syntax constraints the programming language Prolog comes with. However, though the absolute number of required operator declarations is often quite small, the process to develop their precise definitions – both for newly created and existing DSLs – carries with it the complexity of Prolog term parsing. Therefore, it typically requires a deep knowledge of Prolog's term parsing internals, existing operators, their types and precedences, as well as dependencies on each other. Given an example sentence, it is in general difficult to answer the question at hand which can be coined as "Is it *possible* to define it as an internal DSL, and *how*?". The solution to this problem demands an idea which operators have to bind the strongest, and which of them are of prefix, postfix, or infix type, so that the DSL becomes a valid Prolog term or program. It may be the case that this added syntactic weight is counter-productive when defining and working with internal DSLs in Prolog, as it places a burden on the application-domain specialists who would define and use the DSL.

[41] Quote from "93 % of Paint Splatters are Valid Perl Programs" (Colin McMillen, and the fictional second author Tim Toady) from SIGBOVIK 2019. SIGBOVIK is an annual humorous multidisciplinary conference "celebrating the inestimable research work of Harry Q. Bovik", and "specializing in lesser-known areas of academic research". The original paper is available from https://www.famicol.in/sigbovik/. In the experimental setup, most programs were considered valid due to a language feature of Perl that is not commonly present in most other programming languages: Perl supports *unquoted strings*, in which a sequence of alphanumeric characters by itself is parsed as though it were a quoted string. Without whitespace characters, the same applies for Prolog's atoms.

Given example sentences of the considered language and some knowledge about the contained data, our *library(plammar)* is able to infer the required operator definitions with their types, together with the allowed range of possible precedences. After all, the dependencies with respect to priorities that come along with complex term parsing constitute a constraint satisfaction problem over all operator precedences.

Prolog is traditionally strong in solving such combinatorial problems and constraints about integers in finite domains. To reason about the operators' precedences, *library(plammar)* integrates SWI-Prolog's constraint solver CLP(FD). Its background, possible implementations using attributed variables, and results in the context of Prolog source code parsing are presented in Section 10.1.

Besides the declaration of user-defined operators, a second possibility to adapt Prolog for a given application domain is to overcome some of the host language's originally strict syntactic requirements. For instance, SWI-Prolog allows to restrict the notation of named variable tokens to those starting with the underscore character `_`. The first alternative in the EBNF definition of *named variable* in Figure 5.2 can be deactivated by the program flag `var_prefix`[D.7], so that symbols starting with an uppercase letter do not form a variable and are instead treated as an atom. This supports the seamless integration of DSLs that rely on symbols starting with uppercase letters. Similarly, there are various other constraints in the ISO Prolog standard that can be relaxed in order to support a broader variety of DSLs to be integrated internally. Section 10.2 provides an overview and extensive list of possible language extensions for Prolog.

The chapter continues in Section 10.3 with an example application that requires user-defined operators as well as some of the presented language extensions: the query language GraphQL – which we previously integrated in the traditional, external way using DCGs in Chapter 6 – represents valid Prolog terms and thus can alternatively be implemented as an internal Prolog DSL. In Section 10.4, similar considerations are made for XPath expressions that can be part of XML Schema documents. By defining appropriate operators and enabling some language extensions, the XPath expressions can be directly processed as Prolog terms, which allows to refrain from a fully-featured XPath parser in our *library(xsd)*. The results have been used to enhance the XML Schema validator for SWI-Prolog. This application is presented in more detail in our papers [79, 80].

As before, we refer to definitions of the ISO Prolog standard by the abbreviation Iso. In code examples, `s` and `t` denote terms that are known to be valid in Prolog. Normal whitespaces ␣ depict any LTS, whereas · is strictly required to be not empty.

10.1. Operators as a Constraint Satisfaction Problem

In Section 9.4, *library(plammar)*'s approach to combine tokens into Prolog's bigger building blocks, the terms, is introduced. So far, we discussed in detail the parsing process for atomic terms and numbers (ISO 6.3.1), as well as for variables (ISO 6.3.2). Their parsing does not differ from those of similar language constructs in other programming languages. In particular, it does not come with additional difficulties that arise from Prolog's flexible syntax. The same applies for terms written in double quoted list notation enclosed in `"..."` (ISO 6.3.7), as well as for curly bracketed terms (ISO 6.3.6) – their CST creation is made up only of combining tokens and terms together, in the latter case the `open_curly` token, the inner term, and the `close_curly` token, resulting in the overall parse tree for the compound term. Here, the curly bracketed term is valid regardless the inner term's priority.

Compound terms written in functional notation (ISO 6.3.3) are parsed similarly, though the ISO Prolog standard comes with a first restriction regarding the terms that are allowed as the compound term's arguments: according to ISO 6.3.3.1, the nonterminal `arg//0` is either an operator of any precedence, or a term with a priority of less than 1000. The first alternative ensures that the compound term `f(:-, +, /)` is syntactically valid whatever operators are currently defined. The second on the other hand disallows terms like `f(a :- b, c)`, avoiding possible confusion regarding the meaning of the inner comma. It might depict the conjunction within the stated clause in the unary term `f/1`, or alternatively separate the arguments of the binary `f/2`. Therefore, to express terms about clauses, they have to be put into an extra pair of parentheses, as it resets the term's priority, i. e., `f((a :- b, c))` is a valid compound term of functor `f/1` again. Similar considerations are made for compound terms in list notation, as their definition in ISO 6.3.5 refers to the same nonterminal `arg//0` to describe the structure of each list item. Consequently, they have to be terms with a priority from 1 to 999, too.

The previous example underlines the difficulties that arise when parsing Prolog terms. Given the character sequence `f(\+ a)`, it can be a valid Prolog term or not, which depends only on the precedence given to the prefix operator `\+/1`. Following its built-in definition with a precedence of 900 (ISO 6.3.4.4, cf. Table 5.1), no additional parentheses are required. However, users are allowed to reset the operator's precedence to any value, as well as to remove the operator at all by specifying a precedence of 0, which then would make `f(\+ a)` a syntactically invalid Prolog expression.

To make this question more difficult, given only this standalone source code snippet, it is not even clear whether `f/1`'s argument `\+ a` depicts the term `\+(a)` or `a(\+)`,

as similar to `\+/1`, the atom `a/1` could be defined as a postfix operator with a precedence smaller than 1000. What is known for sure is that not both are defined as operators, as otherwise one of them is required to be put in parentheses, as we already discussed for the example term `(\+) / 1` in Section 9.4. However, what happens if `f/1` is also defined as a prefix operator? Would it also be allowed to write `f \+ a` for short?

As it can be seen, a given Prolog source code snippet can be parsed in different ways, depending only on the currently defined operators, and their types and precedences. These operator definitions are required to be known before parsing any Prolog code,[42] thus it can be statically analysed in reasonable time. However, by also supporting the parsing of Prolog terms without knowing the operator definitions in advance, it is possible to answer the question whether a given character sequence *could possibly* constitute a valid Prolog term, and *how*. As a result, given some code fragments of an external DSL, we can figure out if it can also be stated internally.

In this section, we describe an approach which converges from a few examples to a Prolog-friendly syntax which requires no parentheses, using compound terms in operator notation. This is achieved by solving a constraint satisfaction problem (CSP) to determine the precedence and associativity of a set of tokens which are designated as operators, which may subsequently be directly used to parse the DSL. To the best of our knowledge, our *library(plammar)* is the first parser for Prolog that allows to analyse programs and external source code this way, without knowing the exact definitions of the contained Prolog operators in advance.

10.1.1. Motivational Example

As an introductory example, we reconsider the character sequence `f \+ a` and raise the question: Is it possible to define operators so that this is a valid Prolog term? The ISO Prolog standard defines the precedences of operators to be in the range from 1 to 1200. Having `\+/1` as a built-in right-associative (`fy`) operator with a precedence of 900 additionally restricts possible definitions for `f/1` and/or `a/1` as prefix or

[42] In general, parsing Prolog requires the thorough knowledge about all operator definitions within a Prolog software system or package in advance and at compile-time. Although *Part II* of the ISO Prolog standard on Prolog's module system introduces the directives `module/`$\{2, 3\}$ and `use_module/`$\{2, 3\}$, used to declare a module and all its exported or imported operator definitions (cf. Section 3.8), a module's entry points remain unclear. Hence, for practical applications that rely on the parsing of Prolog code of unknown sources – e. g., IDEs, linters, and code formatters –, it is required to analyse the package's call hierarchy to get all applied operator definitions. This is impossible in general, as the module system allows accessing (possibly) internal module files. This again underlines the need for a more flexible Prolog parser that allows to process Prolog programs with some operator definitions being yet unknown. Some Prolog systems, most notably Ciao, use refined module systems to allow for more thorough static analysis [16].

Listing 10.1: Introductory example to restrict a variable `Prec` to finite domains.

```prolog
solution(Prec) :-
  domain(Prec, [600, 601, 602]),
  domain(Prec, [602, 603, 604]).
```

PROLOG

postfix operators. Furthermore, there can be solutions with the inner `\+` not being only a prefix operator, but at the same time also the infix operator `\+/2`.

To answer this in the most general way, we aim for a solution that returns the possible range of values for each operator's precedence. This requires a mechanism that allows to incrementally restrict the possible domain of a variable, starting from the range from 1 to 1200. As long as this domain has not been broken down to a single value, the variable depicting each operator's precedence should remain free.

As introduced for first-order logic in Section 2.2 and for unification in Prolog in Section 3.2.1, a logic variable is either bound or free. The only way to bind a variable to a value – in our case, a term that depicts the remaining domain of the operator's precedence – is by unification. Thereafter, the variable cannot be distinguished from this term. Once the binding is established, it cannot be modified or destroyed anymore. Other variable bindings can be made only by backtracking over alternative solutions in the SLD tree. Using only Prolog's standard predicates, it is not clear how to store *partial* knowledge to a variable `X`, without actually binding it to a single value or a term holding all this information. It simply cannot be bound to a compound term first, and become a number once the remaining domain collapses to a single possible value.

In the traditional, declarative way, the domain of a variable therefore has to be encoded by several first-order predicates. If, e. g., it is known that a solution requires the precedence of an operator to be one of the numbers `600`, `601`, `602`, and at the same time one of `602`, `603`, `604`, this could be written as presented in Listing 10.1. Here, the variable `Prec` depicts the precedence of a particular operator, and the predicate `domain/2` has to be implemented accordingly to handle the encoded relations for a goal `?- domain(?Prec, +List)`. In particular, it should follow these rules:

– Given a set of at least two domains as `domain/2` predicates, it should simplify them. If there is more than one remaining element, `Prec` should be bound to the list of possible solutions.

– If there is only a single remaining solution, i. e., the remaining list has a length of 1, `Prec` should be bound to this value.

– If the given domains are inconsistent, `domain/2` should fail.

Listing 10.2: Implementation of `domain/3` with chaining variables.

```
1  %% domain(+Term, +Ordset, -Domain)                                    PROLOG
2  domain(Prec, Ordset, var_candidates(Prec,Ordset)) :-
3    var(Prec), !,
4    test_for_single(Prec, Ordset).
5  domain(var_candidates(Prec,L0), Ordset, var_candidates(Prec,LN)) :-
6    ord_intersection(L0, Ordset, LN),
7    test_for_single(Prec, LN).
8
9  %% test_for_single(-Var, +List)
10 test_for_single(_, []) :- !, false.       % empty list: inconsistency
11 test_for_single(Prec, [Prec]) :- !.       % single element: bind
12 test_for_single(_, _).                    % otherwise: continue
```

With classical logic variables, this behaviour cannot be achieved. In line 2 of Listing 10.1, the variable `Prec` should be bound to a term that stores the three possible values `600`, `601`, and `602`. On the other hand, after line 3, `Prec` must be bound to the single solution `602`. This is not possible, because `Prec` cannot be both a list (or nested term) and the integer `602`; the SLD resolution mechanism provides no means to check in line 2 if the predicate `domain/2` is called later another time.

10.1.2. Native Implementation with Chaining Variables

A possible way to fix the otherwise illogical reading of Listing 10.1 is to define a predicate `domain/3` instead, which additionally stores the current knowledge about the variable `Prec` in its last argument. It is then used as a chaining variable between the calls of `domain/3`, explicitly passing the state from one call to another:

```
1  solution(Prec0) :-                                                   PROLOG
2    domain(Prec0, [600, 601, 602], Prec1),
3    domain(Prec1, [602, 603, 604], _).
```

The predicate `domain/3` has to be implemented according to the rules specified in Section 10.1.1. A possible implementation is presented in Listing 10.2. Given a free variable as its first argument, the chaining variable is bound to the term `var_list(X, List)` which acts as a storage for its state, with `List` as the ordered list of possible values (ll. 2–4). If on the other hand, the first argument is already bound to a term of this form, the stored list is combined with the new list of possible values (ll. 5–7). This is done with the help of the predicate `ord_intersection/3`[C.9]. It computes the intersection of two given ordered sets and is part of SWI-Prolog's

Listing 10.3: Example queries for `domain/3`, deducing an inconsistency, a single solution, and a remaining list of two elements.

```
?- domain(Prec, [600, 601, 602], D0),                    Toplevel
   domain(D0, [603, 604], D1).
false . % found an inconsistency

?- domain(Prec, [600, 601, 602], D0),
   domain(D0, [601, 602, 603, 604], D1).
D0 = var_candidates(Prec, [600, 601, 602]),
D1 = var_candidates(Prec, [601, 602]) .
% Prec remains free as there are multiple solutions in D1

?- domain(Prec, [600, 601, 602], D0),
   domain(D0, [602, 603, 604], D1).
Prec = 602, D0 = var_candidates(602, [600, 601, 602]),
D1 = var_candidates(602, [602]) .
```

built-in *library(ordsets)* [136, Sec. A.26]. In both cases, the resulting list is tested according to the aforementioned rules to have either no remaining element, only a single one, or more, using the predicate `test_for_single/2` as defined in lines 9–12.

Given these definitions, Prolog is able to recognise an inconsistency in the given domains, to deduce the single solution for `Prec`, or to return an ordered set of possible values as part of the state term bound to the last argument in `domain/3`. Examples for these three cases when using `domain/3` in SWI-Prolog's toplevel are presented in Listing 10.3.

Note that with term expansions it is possible to recognise goals that are called multiple times in a rule's body at compile-time. As a consequence, the original rules of Listing 10.1 can be source-to-source transformed into equivalent ones with chaining variables. However, though feasible, one should not use term expansion to overhaul an otherwise illogical program.

10.1.3. Attributed Variables

With *attributed variables*, it is possible to achieve the same behaviour without extending the original `domain/2` from Listing 10.1 by an additional argument. Instead of storing the information about restricted domains in a special data structure that has to be passed on each call, it can be assigned as an *attribute* to the variable `Prec`. Attributed variables are the basis for constraint-logic programming in SWI-Prolog's *library(clpfd)*, which we rely on in our implementation of operator precedences and

term priorities in *library(plammar)*. We introduce the work with *library(clpfd)* in more detail in Section 10.1.4.

Attributed variables are not part of the ISO Prolog standard. Though they are implemented in all major Prolog systems [133, 134], there is no consensus in the Prolog community on their exact definition and interfaces. Unlike for term expansions, there is no de-facto standard for attributed variables. In our work, we make use of attributed variables as provided by SWI-Prolog. Its interface is identical to the one realised by Bart Demoen for hProlog in 2002 [31]. SICStus Prolog [18, Sec. 10.3] offers predicates of the same names but with slightly different semantics and arguments.

Even though the details differ in the various Prolog systems that implement attributed variables, the predicates they provide share the same idea and can be divided in three kinds, which we introduce in the following.

Predicates for Attribute Manipulation. First of all, we need a predicate for the *addition* of an attribute to a variable. All major Prolog systems implement this in a predicate called `put_attr/`n, though its arity n varies. In SWI-Prolog, an attributed variable is a relation between a variable `Var`, a module `Mod`, and a value `Val`. Therefore, the predicate $\texttt{put\_attr/3}_{\texttt{SWI}}$ takes these three arguments. It does not check for an already set attribute of the same module, but instead simply replaces it.[43] As a consequence, it can also be used to *change* an existing attribute.

For the *retrieval* of an already set value, $\texttt{get\_attr/3}_{\texttt{SWI}}$ realises the same signature for the three arguments `+Var`, `+Mod`, and `-Val`. It unifies `Val` with the currently set value, without deleting it. If `Var` is not an attributed variable, or the named attribute `Mod` is not associated to `Var`, $\texttt{get\_attr/3}_{\texttt{SWI}}$ fails silently.

Unlike the previous predicates, the *removal* of an attribute is not part of all implementations of attributed variables in the various Prolog systems. SWI-Prolog provides $\texttt{del\_attr/2}_{\texttt{SWI}}$ for this purpose. The goal `del_attr(+Var,+Mod)` deletes the attribute named `Mod` for a variable `Var`. If `Var` loses its last set attribute it is transformed back into a traditional Prolog variable. To distinguish a normal Prolog variable from one with set attributes, the predicate $\texttt{attvar/1}_{\texttt{SWI}}$ can be used. It is needed since Prolog's built-in predicate $\texttt{var/1}_{\texttt{ISO}}$ succeeds for all variables, regardless of possibly set attributes.

[43] When backtracked, `put_attr(+Var,+Mod,+Val)` in SWI-Prolog restores the old value. That means that unlike ordinary variables, an attribute is a mutable term, with support for destructive assignments. In our work and the presented introductory example, we do not rely on backtracking the unification of attributed variables.

Predicates to Reason about the Unification. In comparison to traditional logic variables, attributed variables are special only when it comes to unification. Unifying the attributed variable `X` in the module `Mod` with a term `Y` schedules the goal `Mod:attr_unify_hook(+Curr,+Y)`. The built-in predicate `attr_unify_hook/2`$_{\texttt{SWI}}$ can be extended by the user. The user-defined clause is automatically invoked by the Prolog engine at the first possible opportunity (that is why it is called *hook*). Therefore, goals of the form `Mod:attr_unify_hook(·,·)` are not part of the SLD resolution tree as introduced in Section 2.4, but are called before proceeding with the next leaf.

`Curr` is the attribute that was associated to the variable in this module `Mod`, `Y` is any term the attributed variable `X` is unified with. As a result, `Mod:attr_unify_hook(·,·)` is not invoked with the arguments of the unification `=/2`$_{\texttt{ISO}}$, but rather the attribute's value and one term, which could also be another attributed variable. In this case, `attr_unify_hook/2`$_{\texttt{SWI}}$ is typically implemented to combine both attributes and associate the combined attribute with `Y` using `put_attr/3`$_{\texttt{SWI}}$. If the goal `Mod:attr_unify_hook(·,·)` fails, the whole unification fails.

Predicates to Change the Display. By default, SWI-Prolog's toplevel simply prints the attributes of a variable as terms of `put_attr/3`$_{\texttt{SWI}}$:

```
?- put_attr(X, some, 1), put_attr(X, other, 2).                    TOPLEVEL
put_attr(X, some, 1),
put_attr(X, other, 2) .
```

In this regard, it simply repeats the last effective `put_attr/3`$_{\texttt{SWI}}$ goal for each variable `Var` and module `Mod`. This behaviour can be changed by defining the predicate `attribute_goals/3`$_{\texttt{SWI}}$ in the used module. It is automatically invoked by the Prolog engine for each variable with a set attribute that is going to be printed in the toplevel.[44]

Example. Given the previously introduced predicates, the original `domain/2` as used in Listing 10.1 – without an additional third argument – can be shortly defined as presented in Listing 10.4. We store the list of possible solutions in the module `candidates`. Note that this attribute is first defined for a fresh variable `Y` and then unified with the original variable `Prec`, to invoke the unification hook `candidates:attr_unify_hook/2`. It handles the unification of an attributed variable with another term, and as part of this, it also differentiates between the three

[44] `Mod:attribute_goals/3` has three arguments since it is usually defined as the DCG nonterminal `Mod:attribute_goals(+AttVar)//`, with `AttVar` being the attributed variable.

Listing 10.4: Implementation of `domain/2` with attributed variables.

```
1  %% domain(+Term, +Ordset)                                    PROLOG
2  domain(Prec, Ordset) :- put_attr(Y, candidates, Ordset), Prec = Y.
3
4  attr_unify_hook(Domain, Y) :-
5    ( get_attr(Y, candidates, Domain_Y) ->
6      % Y is an attributed variable in the same module
7      ord_intersection(Domain, Domain_Y, New_Domain),
8      put_attr(Y, candidates, New_Domain),
9      test_for_single(Y, New_Domain)
10   ; var(Y) ->
11     % Y is a variable, possibly with attributes in other modules
12     put_attr(Y, candidates, Domain)
13   ; otherwise ->
14     % Y is not a variable
15     ord_memberchk(Y, Domain) ).
16
17 :- op(500, xfy, in).
18 attribute_goals(X, [X in Domain|Rest], Rest) :-
19   get_attr(X, candidates, Domain).
```

cases (cf. Section 10.1.1) when unified with another attributed variable. If two attributed variables of the `candidates` are unified (ll. 5–9), their stored lists of possible values are combined; the result is used as a replacement for `Y`'s attribute (l. 8). We use our previously defined `test_for_single/2` from Listing 10.2 to fail for two disjunctive sets, or bind `Y` in case of a single remaining solution.

If `Y` is a variable without an attribute set in the `candidates` module (ll. 10–12), the attribute is added instead. Given that `Y` is bound (ll. 13–15), it is ensured to be one of the possible remaining solutions with the help of `ord_memberchk/2`[C.10].

In lines 17–19, we use `attribute_goals/3`$_{\text{SWI}}$ to define how to display set attributes in the toplevel. By defining the infix operator `in/2`, the result is nicely printed as in the following example call:

```
?- domain(Prec, [600, 601, 602]),                               TOPLEVEL
   domain(Prec, [601, 602, 603]).
Prec in [601, 602] .
```

10.1.4. Constraint-Logic Programming over Finite Domains

In the previous section, we defined only a single mean to store all possible integer values of an operator's precedence in the form of a list, and to incrementally tighten

Listing 10.5: Basic usage example of *library(clpfd)*.

```
?- use_module(library(clpfd)). % load CLP(FD)                    TOPLEVEL
true .

?- 601 #=< Prec, Prec #=< 603.
Prec in 601..603 .

?- 601 #=< Prec, Prec #=< 603, label([Prec]).
Prec = 601 ;
Prec = 602 ;
Prec = 603 .
```

this domain. This alone is enough to reason about various priorities in a complex Prolog term, because their initial range is restricted by the ISO Prolog standard to the countable values from 1 to 1200. However, since almost all Prolog programs also reason about integers, there is a wide range of well-established existing *constraint solvers* implemented in and for Prolog. For SWI-Prolog, these are *library(clpb)*, *library(clpqr)*, and *library(clpfd)*. Their naming follows the structure *Constraint-Logic Programming over* $\mathcal{X}$, where $\mathcal{X}$ is the targeted application area. The CLP($\mathcal{B}$) module handles Boolean values, CLP($\mathcal{Q}$,$\mathcal{R}$) rational and real numbers, and CLP(FD) is dedicated to integers over finite domains, which happens to suit well for our use case of operator precedences.

The constraint solver CLP(FD) [118] provides predicates for declarative integer arithmetic. They serve as drop-in replacements for Prolog's traditional arithmetic predicates like `is/2`$_{\texttt{ISO}}$, `</2`$_{\texttt{ISO}}$, `=:=/2`$_{\texttt{ISO}}$ and similar, and are for dissociation called *constraints* instead. In our application for *library(plammar)*, we mostly rely on the constraints `#</2`$_{\texttt{SWI}}$ and `#=</2`$_{\texttt{SWI}}$, which have the same meaning as their built-in counterparts without the `#`. Arithmetic expressions are written as before using the classical infix operators like `+/2`$_{\texttt{ISO}}$ and `*/2`$_{\texttt{ISO}}$. In contrast to the logically impure predicates `</2`$_{\texttt{ISO}}$ and `=</2`$_{\texttt{ISO}}$, the constraints constitute relations between their arguments, which therefore can also be both free variables. Consequently, the constraints' instantiation modes are just `(?,?)`.

The implementation of *library(clpfd)* in SWI-Prolog [136, Sec. A.9] is based on attributed variables. The domain of a variable `Prec` is printed along with the computed answer substitution in the toplevel similar to what we did in Listing 10.4, but with the additional infix operator `../2`$_{\texttt{SWI}}$ to denote ranges. Listing 10.5 shows a minimal usage example of *library(clpfd)* in the toplevel. The third goal uses its enumeration predicate `label/1`$_{\texttt{SWI}}$, which takes a list of attributed variables, and binds them systematically to values of their currently set finite domain.

With CLP(FD), we can encode the constraints to the priorities of terms in the definite clause grammars as follows:

```
arg --> { Prec #< 1000 }, term(Prec).
```
PROLOG

This example realises our initially discussed restriction from ISO 6.3.3.1 that arguments of compound terms in functional notation as well as list elements have to be of a priority not greater than 999. Consequently, there is no conflict with the comma, whose precedence of 1000 cannot be changed (ISO 6.3.4.3).

Further constraints are added with respect to the operator's type for compound terms that are written in operator notation. For instance, the infix right-associative operator type `xfy` with precedence N implies that the priority of the first operand is strictly less than N, whereas the priority of the second operand is equal to or less than N. In *library(plammar)*, the constraints for all operator types are modelled in the predicates `prec_constraints/`$\{3, 4\}$, whose definitions are given in Appendix B.10.

10.1.5. Operator Inference in the Library `plammar`

By default, *library(plammar)* takes into account only operators that are given in the ISO Prolog standard, or explicitly defined via `op/3`$_{\text{ISO}}$ directives in the examined Prolog source code. In both cases, the constraints about the operators' precedences behave the same as Prolog's traditional arithmetic predicates, because all variables are known and bound to integer values. Only be providing the option `infer_operators(Os)` in the third argument of `prolog_parsetree/3` or `prolog_ast/3`, the free variable `Os` is bound to a list of possible operator definitions so that the given source code syntactically becomes valid Prolog. The list consists of compound terms of the form `op(?Precedence,?Type,?Name)`, which can be later directly used to declare the operators in a Prolog program via `op/3`$_{\text{ISO}}$ directives. If the variable `Os` is partially bound, it is used as a template, and free variables for the operators' precedences and types are bound as expected.

Solution for the Motivational Example. Listing 10.6 shows an example application for our motivational example `f \+ a`. Since backslashes in double quoted strings need to be escaped, and a valid Prolog program ends the clause with the full stop, we call *library(plammar)*'s `prolog_parsetree/3` with an input of `string("f \\+ a.")`.

In the ISO Prolog standard, `\+/1`$_{\text{ISO}}$ is defined as a prefix operator of type `fy` with a precedence of 900. This built-in operator is taken into account by *library(plammar)*,

Listing 10.6: Calculation of possible operator definitions for the given character sequence `f \+ a.` via `prolog_parsetree/3`.

```
?- prolog_parsetree(string("f \\+ a."), _PT,                    Toplevel
     [infer_operators(Os)]).
% \+(f, a).
Os = [op(B,xfx,\+)], B in 1..1200 ;
Os = [op(B,xfy,\+)], B in 1..1200 ;
Os = [op(B,yfx,\+)], B in 1..1200 ;
% a(\+(f)).
Os = [op(B,xf,\+), op(C,xf,a)], B #<  C, B in 1..1199, C in 2..1200 ;
Os = [op(B,xf,\+), op(C,yf,a)], B #=< C, B in 1..1200, C in 1..1200 ;
Os = [op(B,yf,\+), op(C,xf,a)], B #<  C, B in 1..1199, C in 2..1200 ;
Os = [op(B,yf,\+), op(C,yf,a)], B #=< C, B in 1..1200, C in 1..1200 ;
% f(\+(a)).
Os = [op(A,fx,f)], A in 901..1200 ;
Os = [op(A,fx,f), op(B,fx,\+)], B #<  A, A in 2..1200, B in 1..1199 ;
Os = [op(A,fx,f), op(B,fy,\+)], B #<  A, A in 2..1200, B in 1..1199 ;
Os = [op(A,fy,f)], A in 900..1200 ;
Os = [op(A,fy,f), op(B,fx,\+)], B #=< A, A in 1..1200, B in 1..1200 ;
Os = [op(A,fy,f), op(B,fy,\+)], B #=< A, A in 1..1200, B in 1..1200 .
```

which therefore assumes this operator as given, or explicitly returns it as part of the list `Os`, in case it is required or allowed to be redefined.

Given that the character sequence `f \+ a.` is a valid Prolog program, it could represent the clause `\+(f, a)`, `a(\+(f))`, or `f(\+(a))`, which are all returned by `prolog_parsetree/3` as shown in Listing 10.6. The first variant can be achieved by defining `\+/2` as an infix operator, i. e., it is of the type `xfx`, `xfy`, or `yfx`, which are returned via backtracking.[45] In this case, its precedence `B` can be arbitrarily chosen. The absence of `f` and `a` in the returned list `Os` implies that they are not allowed to be defined as operators as well. If one of them was also an operator, it must be put in parentheses to be valid argument of the infix operator `\+/2`, which does not follow the given character sequence.

The two other alternatives require the inner `\+` to be either a postfix operator (ll. 7–11), or a prefix operator (ll. 12–18). In both cases, the resulting term's principal functor has to be defined as a postfix or prefix operator, too. As a result, the combinations of their types {`xf`,`yf`} and {`fx`,`fy`} results in four computed answer substitutions each. The additional answers in lines 13 and 16 take into account the

[45] Note that the initial declaration of `\+/1` as a prefix operator of type `fy` is not touched, since Iso 6.3.4.3 allows a name to be defined for all operator classes at the same time. The ISO Prolog standard's recommendation to not use the same name as both infix and postfix operator is discussed for the language extension `allow_infix_and_postfix_op`[D.13] again in Section 10.2.3.

definition of `\+/1`$_{\texttt{ISO}}$ from the ISO Prolog standard, and therefore restrict the prefix operator `f/1` to be of type `fx` or `fy` with its precedence chosen accordingly as ≥ 901 or ≥ 900, respectively.

It is worth noting that the operator definitions returned by `prolog_parsetree/3` in Listing 10.6 do not cover the clauses `f(a(\+))` and `a(f(\+))`. They would be returned if `\+/1` was not already defined in the ISO Prolog standard as a prefix operator, thus requiring additional parentheses to be used as an argument for a postfix operator `a/1` or a prefix operator `f/1`. Consequently, for a character sequence `f g a` instead of `f \+ a`, *library(plammar)* returns 19 possible solutions: the eleven from Listing 10.6 without the redundant lines 13 and 16, and additional four for each of the clauses `f(a(g))` and `a(f(g))`.

Valid Tokens Do Not Imply Valid Terms. In Section 9.2.3, we presented the tokenisation of the character sequence `X(1.2.3)`. With *library(plammar)*, we can verify that though this input constitutes valid tokens, they can never be part of a valid Prolog term. This can be tested by calling `prolog_parsetree/3` with the inputs `X(1.2.3)` and `l X(1.2.3) r`. The first tries to parse the character sequence as-is. The second additionally allows to use the input within a larger term with other operators `l` and `r`, which can be of any type and precedence, so that `X` and `(1.2.3)` can be operands of different operators. However, *library(plammar)* always returns `false`, as no operators can be inferred to make the inputs valid Prolog.

The critical part here is `X(`*`T`*`)`, with `T` being any valid Prolog term, including the presented character sequence `1.2.3`, which could represent the compound term `.(1.2, 3)` for an infix operator `./2`. In the parser component of the ISO Prolog standard, whose implementation is presented in Section 9.4, the `open_ct//0` token is either part of a term that is enclosed in parentheses (`open//0` and `close//0`, but also `open_ct//0` and `close//0`), or of a compound term. In the first case, the terms `X` and the enclosed `(T)` are not allowed to follow subsequently without some operator in-between. The second case on the other hand requires the left-hand side to be a `name//0` token.

If-then Rules for Expert Knowledge. In Section 5.4, we define an internal Prolog DSL to express knowledge in the form of if-then rules. With *library(plammar)*, it is possible to infer the required Prolog operators just from the given example sentences, and some known restrictions to the solution. For instance, given the example sentence “if the weather is rainy and there is no umbrella or the weather is a thunderstorm then the clothes are wet”, it is known that words

Listing 10.7: Calculation of possible operator definitions for if-then rules.

```
?- solution(Ops).                                                    Toplevel
Ops = [ op(700,xfx,is), op(700,xfx,are), op(P_a,fx,a),
        op(P_the,fx,the), op(P_no,fx,no), op(P_if,fx,if),
        op(P_then,xfx,then), op(P_and,yfx,and), op(P_or,yfx,or) ],
P_then in 702..1200,
P_if   in 2..699,
P_and  #=< P_or, P_and in 701..1199,
P_or   #< P_then, P_or in 701..1199,
P_the  #< P_if, P_the in 1..698,
P_a    in 1..699 ,
P_no   in 1..699 ; % ...
```

like `weather` and `wet` depict entities in the domain of discourse, and therefore should not be returned as operators. In *library(plammar)*, besides the positive list of `op(Precedence,Type,Name)` terms in the `infer_operators` option, we can also specify a list of things that should not form operators in the `disallow_operators` option. This way, `disallow_operators([op(_,_,weather), op(_,yf,_)])` avoids the generation of `weather` as an operator of any type, as well as solutions that rely on postfix operators of type `yf`.

The full source code to generate possible operator definitions so that the aforementioned example sentence becomes valid Prolog is given in Appendix B.11. Its first computed answer substitution is shown in Listing 10.7. It can be further restricted by providing additional CLP(FD) constraints. For instance, to reflect that conjunction via `and/2` binds stronger than the disjunction `or/2`, we can add the constraint `P_and #< P_or`, which incrementally tightens the range of precedence values for `P_and`. Our manually created operator definitions for if-then rules from Table 5.3 is an instance of the returned operator precedences.

10.2. Prolog Language Extensions

When it comes to the internal integration of DSLs, there are several options that allow relaxing the syntactic requirements for Prolog programs. In general, the ISO Prolog standard as well as our Prolog parser implementation in *library(plammar)* as presented in Chapter 9 offers two facilities for extending Prolog's syntax or for defining deviations. Firstly, both can be related to the *lexer* component, i. e., we can change what forms a valid token in Prolog (cf. Sections 9.2 and 9.3, Iso 6.4 & 6.5). The second option is a customisation of the ISO Prolog standard's rules what tokens form valid Prolog terms in the *parser* component (cf. Section 9.4, Iso 6.2 & 6.3).

Listing 10.8: Internal Prolog DSL to express logic formulas.

```
:- set_prolog_flag(allow_variable_name_as_functor, true).  SWI-Prolog
:- op(1000, xfy, ∧).
:- op(1100, xfy, ∨).
:- op(1200, xfx, ←).

Path(a, c) ← Edge(a, c) ∨ Edge(a, b) ∧ Path(b, c).
```

Because almost all character sequences already constitute valid sequences of Prolog tokens, usually the latter approach is preferred to define new language constructs in a backwards compatible way. Changes on the tokenisation level on the other hand often just extend the language's alphabet by additionally allowed characters.

10.2.1. Motivational Examples

As a motivational example, Listing 10.8 applies both techniques for language extensions together with user-defined Prolog operators. It allows to express logic formulas with their connectives verbatim as an internal domain-specific language in SWI-Prolog. Following the ISO Prolog standard, Prolog's alphabet is restricted to the 7-bit US-ASCII character set. SWI-Prolog expands this constraint to all Unicode characters, including for the definition as operators. Additionally, the language extension `allow_variable_name_as_functor`[D.21] allows to use symbols starting with an uppercase letter as functors in compound terms. It is similarly supported by both our *library(plammar)* and recent versions of SWI-Prolog. Only because of these extensions, domain experts are able to state logic formulas as Prolog clauses using the traditional notation as illustrated in Listing 10.8, where `Path` and `Edge` are predicates, and `←`, `∨` and `∧` denote implication, disjunction, and conjunction, respectively. Though syntactically `a`, `b` and `c` are Prolog atoms as well, their intended meaning of logic variables can be realised at compile-time by an appropriate term expansion (cf. Section 5.2).

Listing 10.9 provides a second example for a DSL that can be integrated internally. The definition of the predicate `print_list_of_lists` closely resembles an imperative programming language, but is an executable SWI-Prolog program as well. In Sections 10.2.2 and 10.2.3, we discuss the language extensions that are required to successfully and unambiguously parse it as valid Prolog code. There, we restrict ourselves to options of *library(plammar)* which have been already mentioned in the two motivational examples, or are referenced by the internal integrations of GraphQL and XPath in Sections 10.3 and 10.4.

Listing 10.9: Exemplary internal Prolog DSL with language extensions that are also supported by SWI-Prolog.

```
:- set_prolog_flag(allow_variable_name_as_functor, true).   SWI-Prolog
:- op(100, xf, {}).
:- op(100, xf, []).
:- op(500, xf, ;).
:- op(600, xfx, ∈).
:- op(600, fx, while), op(500, fx, function).

function print_list_of_lists() {
  while (X ∈ [[a, b], [1, 2, 3]]) {
    writeln(LENGTH(X));
    writeln(X[0]);
  }
}.
```

The complete index of all language extensions that we propose and that can be used with *library(plammar)* is given in Appendix D. The index is divided regarding the extension's application in the overall parsing process, i. e., whether they apply on the tokenisation or parser level. It might serve as a reference, since for each language extension we additionally provide a source code example that is invalid according to the ISO Prolog standard but valid after enabling the discussed flag.

In *library(plammar)*, each extension can be enabled separately in the list of options that is passed via the third argument in the predicates `prolog_{tokens,parsetree,ast}/3`. Some of the proposed language extensions were originally introduced by SWI-Prolog and are therefore already supported by its recent versions. These cases are indicated in Appendix D, and *library(plammar)*'s option name is the same as used by SWI-Prolog's *environment flags* [136, Sec. 2.12], like for instance the aforementioned `allow_variable_name_as_functor`[D.21] in Listings 10.8 and 10.9. In contrast to our implementation, which aims to be compatible with many different Prolog systems and dialects and thus requires delicate distinctions of features, some of SWI-Prolog's deviations from the ISO Prolog standard cannot be disabled separately or at all.

10.2.2. Tokenisation Level

Language extensions for Prolog on the tokenisation level either target the *alphabet*, or they declare *refined tokens* or *new tokens*. For each of these three areas, we present a selection of extensions that are supported by *library(plammar)* in this section and elaborate on their advantages for the integration of external languages. The complete

list is given in Appendices D.1 to D.12. Most of these extensions on the tokenisation level have their origin in deviations from the ISO Prolog standard by some popular Prolog system. They are covered by *library(plammar)* and have proven to be particularly useful when parsing Prolog programs of various sources, as performed in our empirical community evaluation of Prolog programming styles in [82].

Alphabet. The first class of extensions on the tokenisation level is oriented towards the language's alphabet, i. e., the set of characters valid Prolog programs consist of. Extensions expand the countably infinite set of symbols that can be used to form a token (cf. Section 3.1).

For instance, the flag `allow_unicode`[D.1] enables the literal usage of Unicode characters. Traditionally, Prolog's alphabet is restricted to the 7-bit US-ASCII character set, thus symbols like `∈` cannot be used anywhere in the Prolog source code. As a consequence, `∈` does not constitute a valid operator name, neither can it be stated as part of source code annotations. For the integration of external DSLs that rely on Unicode characters, the setting `allow_unicode(true)` enables the full Unicode character set in *library(plammar)*.

Further language extensions of this class are provided by the options `allow_symbolic_no_output_char_c`[D.4], `allow_symbolic_escape_char_e`[D.4], and `allow_symbolic_space_char_s`[D.4], which add the originally missing symbolic control characters `\c`, `\e`, and `\s` from the 7-bit US-ASCII set to Prolog's alphabet.

Refined Tokens. Only a minor part of extensions on the tokenisation level affect the language's set of allowed characters. In contrast, most extensions combine or restrict tokens that are already allowed by the ISO Prolog standard. Because Prolog does not permit two operands to be adjacent, any two Prolog tokens that are consecutively written in a DSL's source code and of which neither is declared as an operator can never constitute a valid Prolog term. We previously discussed this for the input `X(1.2.3)` in Section 10.1.5. Consequently, the sequence of such tokens can be defined as a new, combined token. Other Prolog language extensions then again provide restrictions to the tokens as defined in the ISO Prolog standard to change their meanings.

One example for a refined token has been introduced in Section 9.3.4 when discussing the efficient parsing of numbers: SWI-Prolog allows to split long integers into digit groups, separating them either by single spaces or underscores. This can be enabled in *library(plammar)* by the two options `allow_digit_groups_with_space`[D.8]

and `allow_digit_groups_with_underscore`[D.9]. Since separating large numeric literals into digit groups by underscores is supported by several other programming languages, this Prolog language extension supports their integration as internal DSLs. Further compatibility with representations of numbers from other languages is provided by the extensions `allow_integer_exponential_notation`[D.10] and `rational_syntax`[D.11].

As introduced in Section 3.1.1 and formally specified in the EBNF definition of the nonterminal *variable* in Figure 5.2 of Section 5.6, a Prolog variable starts either by an uppercase letter, or by the underscore as its prefix. By enabling the setting `var_prefix`[D.7], only the second form denotes valid variable names. Symbols beginning with an uppercase letter are then treated as atoms instead. Although this language extension does not expand Prolog's expressiveness, it contributes to the integration of case-sensitive external languages as internal Prolog DSLs since we work around the possibly unintended special meaning of logic variables.

New Tokens. In a third form of language extensions on the tokenisation level, we define new tokens that otherwise cannot be parsed at all, although all of their characters are part of Prolog's alphabet.

For instance, the input `\u2208` neither constitutes a single token nor a sequence of tokens according to the ISO Prolog standard. It can be accepted as valid Prolog source code only after enabling the setting `allow_unicode_character_escape`[D.2]. This language extension adds support for the de-facto standard known from other programming languages to encode Unicode characters like `∈`, which is of hexadecimal number 2208. The following flags constitute similar extensions:

- `allow_missing_closing_backslash_in_character_escape`[D.3]: `\xa`
- `back_quoted_text`[D.6]: `` `example` ``
- `allow_single_quote_char_in_character_code_constant`[D.12]: `0''`
- `allow_newline_as_quote_char`[D.12]: `0'<newline>`
- `allow_tab_as_quote_char`[D.12]: `0'<tab>`

In many programming languages, it is possible to specify the program loader script path at the file's beginning. This character sequence is started by `#!` (called *shebang* or *hashbang*). Due to its special meaning, it is allowed only in the program's very first line of code. In our implementation of the Prolog lexer as introduced in Section 9.3.2, it thus appears in the finite-state machine as (*sh*) directly after the initial state (*pl*). Allowing the shebang via the setting `allow_shebang`[D.5] simplifies the processing of

source code from another programming language. Its support has to be realised at the tokenisation level, since the arbitrary character sequence that follows `#!` is not required to be a valid Prolog term, thus just defining both `#/1` and `!/1` as prefix operators does not suffice.

10.2.3. From Tokens to Valid Terms

There are many sequences of tokens that do not constitute valid Prolog terms. As introduced in the previous section, these cases offer the chance to extend the ISO Prolog standard. In addition to *refined tokens*, this can also be achieved on the term parsing level, i. e., by extending Prolog's definition of what tokens constitute a term. Extensions of this form are guaranteed to be backwards compatible, since they only add new language constructs and do not modify the existing. Their implementation expands Prolog's strength in modelling languages as internal DSLs.

In this section, we introduce a selection of language extensions on the term parsing level. The complete list with detailed information is given in Appendices D.13 to D.24. In the following, we restrict ourselves to the motivational example of Listing 10.9. It combines some of the proposed extensions to internally integrate a DSL that resembles an imperative scripting language. In the given form, it is already supported by recent versions of SWI-Prolog. We shortly describe each applied extension with respect to its corresponding line of code:

(1) By using the flag `allow_variable_name_as_functor`[D.21], the functors of compound terms are allowed to start by an uppercase letter. This way, `LENGTH(X)` in line 10 of Listing 10.9 is read in as if it were written as `'LENGTH'(X)`. This extension was originally suggested by Robert van Engelen and is supported by SWI-Prolog since at least version 2.8 from 1990 [127].

(2) Without further modifications, both `{ }` and `{ t }` are valid Prolog terms according to the ISO Prolog standard. Seen individually, the empty curly brackets of line 2 constitutes a valid atom (ISO 6.3.1.3). However, curly brackets are used in most imperative programming languages to enclose code blocks. With the extension `allow_curly_block_op`[D.16], it is allowed to use the atom `{}` as an operator of any type. With the presented operator definition, the curly bracketed term of lines 8–13 is treated as a postfix operator with the argument `function print_list_of_lists()`.

(3) Besides curly bracketed terms to enclose code blocks, imperative programming languages often use postfix terms in square brackets to denote ar-

ray subscripting as in line 11. This can be similarly enabled by the flag `allow_square_block_op`[D.17].

(4) ISO 6.3.4.3 recommends to not declare the same name as an infix and postfix operator. This is justified by possible optimisations for the parser, though several systems like SWI-Prolog and SICStus Prolog lift this restriction. In *library(plammar)*, this extension can be enabled by the flag `allow_infix_and_postfix_op`[D.13]. It is required to terminate the second statement in line 11 by `;`. In contrast, `;/2`$_{\texttt{ISO}}$ of the previous line represents the infix operator as defined by the ISO Prolog standard.

(5) As introduced in Section 10.2.2, the flag `allow_unicode`[D.1] is required to be able to define `∈/2` as an infix operator. It is later used in line 9.

(8) In line 6, we define the keywords `while/2` and `function/1` as prefix operators. Consequently, `print_list_of_lists()` is the operand of `{}/1`. It is a valid Prolog term only by the flag `allow_compounds_with_zero_arguments`[D.14], which allows to use compound terms with an empty arguments list.

Note that similar to operator definitions, our *library(plammar)* is able to automatically infer the required settings to accept an otherwise invalid Prolog program. This is because we allow all arguments of the predicates `prolog_tokens/3`, `prolog_parsetree/3`, and `prolog_ast/3` to be free or only partially bound variables, in particular the third argument that represents the list of options. Consequently, we can infer the required Prolog language extensions just by example sentences of the external DSL. This finally answers the question *if* it is possible to integrate it internally, *how*, and *on which Prolog systems*, since the proposed language extensions are variously supported by existing implementations.

10.3. GraphQL as an Internal DSL

In Section 6.4, we present the integration of the application layer query language GraphQL into Prolog. The resulting framework implements the GraphQL type system and query syntax as an external DSL in Prolog. Instances of GraphQL's formally specified type system definition language can be stated without further modifications in quasi-quotations, which are transformed via DCGs into corresponding Prolog data representations based on dicts.

This traditional approach to integrate an external DSL is powerful yet complex. As argued in our work [84, Sec. 3], it is required because some elements of the GraphQL syntax conflict with those of Prolog – at least if one aims for full compliance with the

Listing 10.10: GraphQL type definitions from Listing 6.8 as internal Prolog DSL.

GraphQL DSL

```
1  :- set_prolog_flag(var_prefix[D.7], true).
2  :- op(100, fx, type).
3  :- op(600, xfy, :).
4  :- op(500, xf, !).    % allow_operator_as_operand[D.19]
5  :- op(400, xfy, {}).  % allow_curly_block_op[D.16]
6  :- op(200, xf, {}).   % allow_infix_and_postfix_op[D.13]
7
8  schema(_schema) :-
9    _schema =
10     type Query {
11       person(name: String!): Person,
12       book(title: String!):  Book,
13       books(filter: String): [Book]
14     }
15
16     type Book {
17       title:   String!,
18       authors: [Person]
19     }
20
21     type Person {
22       name:    String!,
23       age:     Integer,
24       books(favourite: Boolean): [Book],
25       friends: [Person]
26     }.
```

ISO Prolog standard. By lowering some of Prolog's syntax requirements as proposed in Section 10.2, GraphQL's type system can alternatively be specified as an internal Prolog DSL. Listing 10.10 presents the required operator declarations to state the exemplary GraphQL type definitions from Listing 6.8 internally as Prolog terms in the clause `schema/1`.

In line 3 of Listing 10.10, we declare `:/2` as an infix operator, so GraphQL's key-value pairs become syntactically valid. Although the type `xfx` would be more appropriate to avoid chaining, our definition follows SWI-Prolog, which uses the operator $\texttt{:/2}_{\texttt{SWI}}$ to explicitly qualify modules, and in the syntax of dicts. The declaration of `!/1` as a postfix operator allows the specification of non-nullable GraphQL types. The predicate $\texttt{!/0}_{\texttt{ISO}}$ is frequently used in Prolog to cut solutions in the SLD resolution. Being additionally defined as an operator normally entails that calls have to be wrapped in parentheses (Iso 6.3.1.3), which might conflict with existing programs. This can be avoided by the language extension `allow_operator_as_operand`[D.19].

The curly block operator `{}` finally enables the statement of the key-value lists, but in contrast to dicts allows a LTS after `Query`, `Book`, and `Person`. With the option `allow_infix_and_postfix_op`[D.13], it has been defined both as postfix and infix operator to avoid commas between the `type` blocks.

The canonical representation of the resulting compound term can be printed using the built-in predicate `write_canonical/1`:

```
?- schema(_s), write_canonical(_s).                                    TOPLEVEL
{}({ ','(:(person(:(name, !(String))), Person),
         ','(:(book(:(title, !(String))), Book), ...
```

Note that unlike in our implementation with quasi-quotations in Section 6.4, the types like `String` and `Person` denote atoms and not variables, i. e., their references have to be implemented by the processing meta-interpreter or term expansions.

The clause from lines 8–26 of Listing 10.10 can also be used as an example input for *library(plammar)* to automatically infer the required operators to constitute a valid Prolog program. It similarly calculates the possible operator definitions for `:`, `!`, `type`, and the dependencies regarding their precedences and types. With respect to the required language extensions, *library(plammar)*'s first solutions contain only the flags `allow_curly_block_op`[D.16] and `allow_infix_and_postfix_op`[D.13]. To be valid Prolog, it is not relevant whether the elements that start with an uppercase letter – `Query`, `String`, and the other types – represent variables or atoms. This is because at any position in a valid Prolog program where a variable is allowed, an atom could be equally stated, even without any language extension. On the other hand, by explicitly giving the flag `var_prefix`[D.7] as an option to *library(plammar)*, many additional term representations are returned, since the former variables like `Query` then can also constitute operators. Therefore, for inputs where symbols starting with an uppercase letter should be handled as atoms but do not constitute operators, we suggest to either run *library(plammar)* with an explicit `var_prefix(false)` flag, or by passing these atoms to the `disallow_operators` list (cf. Section 10.1.5).

10.4. XPath Expressions in Prolog

The *extensible markup language* (XML) [11] is one of the most used data formats to store and exchange structured data. In particular in the context of web services, XML documents are often used for data transfer and as configuration files. These use cases emphasise the importance for tools that guarantee an expected format of the used XML documents.

A typical approach to specify the structure of an XML document is to formally define its schema as a finite set of allowed attributes, elements, and primitive data types. In its simplest form, this can be achieved by a *document type definition* (DTD). More expressive languages like the *Relax NG Schema* [119] and the *XML Schema Definition* (XSD) [35] additionally allow to specify very detailed constraints on the structure and content of XML nodes.

SWI-Prolog traditionally has a good support for XML. This originates from its tight connections to the emerging research field of semantic data, for instance with SWI-Prolog's semantic web framework *Cliopatria* [132]. It uses RDF/XML [6] as its primary data exchange format, which again is an application of XML and thus depends on stable and fast parsers and schema validators. As part of our contribution, we developed the XML Schema validator for SWI-Prolog, *library(xsd)*. It contains student contributions from Jona Kalkus, Kevin Jonscher, and Lucas Kinne. *library(xsd)* is published under MIT License at https://github.com/fnogatz/xsd. It is available for installation from SWI-Prolog's list of add-ons and can thus be conveniently installed via `?- pack_install(xsd)`. The module ships with more than 1000 tests using our *library(tap)*.

With *library(xsd)*, documents in XML and XSD can be directly embedded via quasi-quotations into Prolog source code. The code snippets are therefore parsed and processed like an external DSL (cf. Chapter 6). In the validation process, the XML nodes are flattened into plain old Prolog facts, following the author's previous work with *xsd2json*.[46] In the validation process, both the generated syntax trees of the XML and XSD documents are traversed simultaneously. For more details on the architecture and implementation of *library(xsd)*, we refer to our work [79, 80].

While the overall XSD documents are integrated externally, the possibly contained XPath expressions are in contrast treated as native Prolog terms in our *library(xsd)*. XPath is an important part of XSD since version 1.1 became a W3C Recommendation in 2012 [44], as it heavily relies on it for the specification of assertions and conditional type assignments. Although completely backwards compatible, these new XSD features require the handling of expressive, declarative rules which can often not be easily added to existing tools. As a result, the number of XSD validators with support for the most recent XSD 1.1 standard is still limited.[47] Processing the contained XPath expressions just as normal Prolog terms in *library(xsd)* allows us to

[46] *xsd2json* is a standalone tool that translates an XSD into an equivalent JSON Schema using Prolog and Constraint Handling Rules (CHR). It is published under MIT License at https://github.com/fnogatz/xsd2json. A detailed system description is given in our paper [77].

[47] Three of the most popular tools with support for XSD 1.1 are *Apache Xerces2 Java* (https://xerces.apache.org, Apache License 2.0), *Oxygen XML Editor* (https://www.oxygenxml.com, proprietary license), and *Saxon XSLT* (https://www.saxonica.com, proprietary license).

Listing 10.11: Examples for XPath expressions that are embedded in an XML Schema of version 1.1. The complete XSD is given together with a conforming XML document in Appendix B.12.

XML Schema

```
① <xs:restriction base="xs:date">
    <xs:assertion test="fn:day-from-date($value) eq 24" />
  </xs:restriction>

② <xs:complexType>
    <xs:assert test="entry[@id mod 2 eq 0]/@class = 'even' or
                     entry[@id mod 2 eq 1]/@class = 'odd'" />
    <xs:assert test="every $i in 1 to count(entry)-1 satisfies
                       (entry[$i] lt entry[$i+1])" />
  </xs:complexType>

③ <xs:element name="entry" maxOccurs="unbounded" type="entry">
    <xs:alternative test="@class = 'even'" type="entry-even" />
  </xs:element>

④ <xs:key name="entry-id-key"><!-- similar for xs:unique -->
    <xs:selector xpath="entry" />
    <xs:field xpath="@id" />
  </xs:key>
```

implement the corresponding rules for the XML Schema validation as normal Prolog clauses.

Listing 10.11 provides examples for each of the XSD 1.1 elements that rely on XPath expressions to specify constraints: `xs:assertion`, `cx:assert`, `xs:alternative`, `xs:key` and `xs:unique`. It is an extract of the XSD document that is given in Appendix B.12. There, we additionally provide an XML document that conforms to the embedded constraints:

① `fn:day-from-date($value) eq 24` is a restriction to the simple type `xs:date`. It is fulfilled only for dates with a day component of 24, as in `1993-05-24`.

② Similarly, constraints for complex types can be stated in `xs:assert` nodes. The first example ensures that every XML node `<entry>` with an even-numbered attribute `id` has the `class` attribute set to `even`, and similar for odd-numbered values of `id`. The second `xs:assert` is true if the values given in the `<entry>` nodes are sorted in ascending order.

③ Type alternatives can be specified as XPath expressions in `xs:alternative` nodes. Here, the XSD type `entry-even` is used for all `<entry>` nodes with their

`class` attribute set to `even`. Otherwise, the XML node is validated against the type `entry`.

④ Only for completeness we also mention `xs:key` nodes here. Their selectors are based on a small subset of XPath. Restrictions via `xs:key` and `xs:unique` are widely used, since they are part of the older XSD 1.0 standard. In the shown example, all `id` attributes of the `<entry>` nodes have to be unique.

In our *library(xsd)*, we use SWI-Prolog's built-in parser to process the embedded XPath expressions as normal Prolog terms. Given the necessary operator definitions that constitute this internal DSL, they are correctly parsed by the ISO Prolog standard's predicate `read_term/2`$_{\text{ISO}}$. In addition, the Prolog program that validates a given XML document against the embedded XPath expressions can be elegantly written using the same operators. For instance, the evaluation of arithmetic expressions is specified in clauses of this form:

```
:- op(700, xfx, [lt, in]).                                          PROLOG
xpath_expr(Context, Value1 lt Value2, Result) :- % ...
xpath_expr(Context, Var in Generator, Result) :- % ... and similar
```

The complete list of operators required for XPath as an internal DSL is given in Appendix B.13. With `every/1` and `$/1` as prefix operators, `to/2`, `satisfies/2`, `lt/2` and `in/2` as infix operators, and finally the square brackets `[]/1` as postfix operator (cf. Section 10.2.3), the second XPath expression in ② is read as the following Prolog term in canonical form:

```
?- read_term(XPath,[]), write_canonical(XPath).                     TOPLEVEL
|: every $i in 1 to count(entry)-1 satisfies (entry[$i] lt entry[$i+1]).
satisfies(every(in($(i), to(1, -(count(entry), 1))),
            lt([]([$(i)], entry),
               []([+($(i), 1)], entry)))
XPath = (every $i in ...). % toplevel prints binding of XPath
```

In addition to `allow_square_block_op`[D.17], the internal integration of XPath depends on a second language extension for Prolog. This is due to the frequent use of the character sequence `()` to denote an empty sequence. The ISO Prolog standard defines `{`, `}`, `[`, and `]` as dedicated tokens. Their pairs `{ }` and `[ ]` (with optional LTS in-between) constitute atoms according to ISO 6.3.1.3. However, an equivalent term for the empty pair of parentheses `( )` is missing in the ISO Prolog standard. We thus propose the language extension `allow_empty_atom`[D.15], which adds to the ISO Prolog standard's EBNF (cf. Section 9.2) the following two grammar rules:

```
atom = open, close ;                                    Ebnf Dsl
atom = open_ct, close ;
```

We also would like to emphasise that Wielemaker et al. noted the need for such a language feature in [130].

11

Conclusion

The worst thing that happened to logic programming is that we stuck with Prolog.

— Statement in the discussions at *Declare 2017*[48]

In this work work, we have provided a number of contributions regarding the definition and application of domain-specific languages in Prolog as well as the efficient implementation of Prolog parsers. Furthermore, we proposed and discussed several extensions that expand Prolog's expressiveness and compatibility among various systems. For a summary of each chapter, we refer to Section 1.4.1.

In Section 11.1, we present empirical results of using *library(plammar)* to statically analyse Prolog source from packages provided by the community and shipped with SWI-Prolog. It continues our summary of the thesis' main results and their implications on the field of research from Section 1.3. This section also emphasises the usefulness and practical applications of the created toolchain and the internally integrated Prolog DSLs, and provides insights on the current dissemination of the proposed language extensions in the Prolog community. Possible avenues for further improvements and research are given in Section 11.2.

11.1. Empirical Results

To the best of our knowledge, the package *library(plammar)* is the first parser for Prolog that allows to automatically infer operator definitions and required flags from example sentences. It has many practical applications and improves the work with Prolog in general, as well as the definition and implementation of domain-specific languages in Prolog in particular.

[48] Quote at the *Declare 2017 Conference and Summer School on Declarative Programming* at the University of Würzburg in September 2017, at which the author was one of the local organisers. In the community discussions about the status quo of logic programming, an attendee noticed that sticking to the traditional syntax and semantics of Prolog would block the progress and population growth of the logic programming paradigm in its entirety. Quote taken from https://twitter.com/ulmerleben/status/910495526168203264.

Parse Tree Generation and Serialisation with Sequences. The package *library(plammar)* is based on *library(dcg4pt)*. Though the idea of an additional argument to hold parse trees in definite clause grammars has been examined before, we extend the classical considerations for natural language processing by sequences of nonterminals of arbitrary numbers, which is a common requirement for parsing and serialising formal language. In addition, the underlying term expansion is agnostic to the mode the DCG is called with. Depending on whether the list argument and/or parse tree argument is bound, it either greedily consumes or produces elements, or starts with the smallest sequence first.

Inference of Operators and Flags. In our work [82, Sec. 4.1], we evaluate the status of and adherence to coding guidelines in the Prolog community. It requires to parse Prolog source code of various sources. In large Prolog codebases, the declarations of user-defined operators as well as the set program flags are usually split across multiple files. To be still able to process a large number of Prolog files independently and in parallel, the used *library(plammar)* has to infer most of the operators and used flags. All in all, the considered codebase consists of more than 4500 Prolog files from 251 packages of the community and 34 packages that are shipped with SWI-Prolog. With a fixed timeout set to 10 seconds, more than 90 % of all Prolog files were successfully parsed by *library(plammar)*. Their sizes reach up to 1 MB with 20.000 lines of Prolog code.[49]

Dissemination of Proposed Language Extensions. Some of our proposed extensions for Prolog are already available in recent versions of SWI-Prolog, using the corresponding environment flags. In the aforementioned empirical study, we also use *library(plammar)* to evaluate the distribution in the Prolog programming community of these. After all, most features are not widespread yet. One reason is that some syntactic elements are situational, e. g., the usage of the shebang `#!` to specify the program's loader script path (`allow_shebang`[D.5]). For others, there might be suitable alternatives. For instance, the option `allow_integer_exponential_notation`[D.10] becomes obsolete once the traditional exponential notation with a leading fractional part is consistently used. Additionally, these extensions were introduced fairly recent into SWI-Prolog while many of the more than 280 examined packages already exist significantly longer. Thus, it is quite unlikely that existing code is rewritten only to use new language features.

However, `dicts`[D.18] seem to be an exception regarding their increasing adoption in the Prolog programming community. Around 14 % of the 251 examined community

[49] A list of all tested packages with their parsing output from *library(plammar)* is available at https://github.com/fnogatz/plammar-community-evaluation.

packages and 25 % of those that are additionally shipped with SWI-Prolog make use of dicts at some point. This is an indicator of both active development and adaption of new language features, as well as of the interest in the community to use proper associative data structures.

Toolchain for Program Transformations. In [85], we describe how to use *library(dcg4pt)* and *library(plammar)* for transformations on Prolog programs. All three predicates `prolog_tokens/3`, `prolog_parsetree/3`, and `prolog_ast/3` are logically pure, so they can be used for the parsing of Prolog programs and their serialisation as well. Since the question of how to concretely format the clauses encoded in an AST is underdetermined, we define sensible code formatting rules in *library(plammar)*. They are based on the coding guidelines given by Covington et al. [25].

Internal Integration of DSLs. The presented internal integrations simplifies the work with popular domain-specific languages in Prolog. The definition of EBNF is used by *library(plammar)* and illustrates the strengths of the presented approach to define DSLs internally: the formal language specification given in the ISO Prolog standard immediately constitutes an executable Prolog program as well. Similar observations can be made with GraphQL. Given the type system in a well-defined format, which was slightly adapted to be compatible with Prolog, our *library(graphql)* creates the corresponding application layer. And finally the definition of XPath as an internal Prolog DSL significantly reduces the programming overhead, as the application of selectors and assertions is implemented in *library(xsd)* simply as a tree traversal over Prolog data structures.

11.2. Future Work

In the future, we need to investigate two main directions. Firstly, the current toolchain with *library(dcg4pt)* and *library(plammar)* can be improved and optimised, in particular regarding its performance as well as the code's maintainability and reusability. Secondly, we can examine further extensions to Prolog and emphasise their usefulness for the integration of DSLs in the logic programming community. In the following, we discuss some of this future work in more detail.

Further Applications with *library(plammar)*. In Section 9.1.1, we already motivated several applications for our *library(plammar)* and emphasised how they profit

from a fully-featured Prolog parser written in Prolog. The greatest potential probably lies in tools that assist with the development of Prolog programs and internal DSLs. Because of its flexible syntax with user-defined operators, integrated development environments (IDEs) are usually missing typical features like syntax highlighting, code completion, and debugging tools when it comes to Prolog. A possible approach to overcome this limitation is to implement the standardised *language server protocol* (LSP) [13, 101] on top of *library(plammar)*. The LSP constitutes a unified API that is supported by most modern development tools. Current integrations of LSP with Prolog make use of SWI-Prolog's introspection capabilities, and SWI-specific predicates for code formatting, e. g., `portray_clause/2`$_{\texttt{SWI}}$.[50] As a result, they return useful insights only for valid Prolog programs. An implementation based on *library(plammar)* on the other hand enables a more forgiving parsing experience and thus provides valuable feedback during the process of development.

In Section 9.5, we describe the architecture of a tool that reformats Prolog source code and checks the adherence to coding conventions using *library(plammar)*. Though this already returns promising results for Prolog programs of any kind, we intend to broaden the scope of language features our automatic code formatter and linter can be applied to. In particular, it could provide specialised formatting rules for internal Prolog DSLs. For instance, though they are among the most widely used Prolog DSLs, the Prolog community has not yet decided on how to consistently format rules in CHR (cf. Section 4.3.3.1), and how to best state constraints in CLP(FD) (cf. Section 10.1.4).

Improve Run-Time Performance. As illustrated by the results of our empirical study in Section 11.1, the overall run-time performance of *library(dcg4pt)* and *library(plammar)* is already promising and allows to use them to parse and serialise fairly large Prolog programs. However, we still see potentials for further improvements. First of all, the work with formal languages that contain sequences of arbitrary number sometimes results in consecutive evaluations of identical goals – already inferred knowledge gets discarded because of backtracking alternatives, even though there are some subgoals which might occur identically in later computations again. A similar problem has been observed before in our XML Schema validator *library(xsd)* [79, Sec. 3]. The resulting performance issues can be resolved by memoisation techniques, e. g., using SWI-Prolog's *tabled execution* [136, Sec. 7] with the `table/1`$_{\texttt{SWI}}$ directive.

[50] Two existing implementations of the LSP for Prolog are the *Language Server Protocol server for SWI-Prolog* (https://github.com/jamesnvc/lsp_server, BSD License) by James Cash, and the *VIM-Plugin for Prolog* (https://github.com/LukasLeppich/prolog-vim, MIT License) by Lukas Leppich.

Listing 11.1: Implementation of `prolog_tokens/3` with a varying execution order of the goals in the rule body.

```prolog
prolog_tokens(string(String), Tokens, Options) :-
  !,
  I0 = string_chars(String, Chars),
  I1 = prolog_tokens(chars(Chars), Tokens, Options),
  ( nonvar(String) -> Instructions = (I0, I1)
  ; Instructions = (I1, I0) ),
  call(Instructions).
```

Another performance bottleneck in *library(plammar)* is the use of SWI-Prolog's *library(option)* to pass and return the required operator definitions and language extensions in the third argument of the predicates `prolog_tokens/3`, `prolog_parsetree/3`, and `prolog_ast/3`. The open lists easily get very long for large Prolog programs. Because of Prolog's implementation in the form of linked lists and the lack of data structures with random access, this creates an overhead in the parsing process that is linear to the number of operators. Again, memoisation via tabling or the use of associative lists via dicts or SWI-Prolog's *library(assoc)* [136, Sec. A.4] could improve the parsing performance of our *library(plammar)*.

(Automatically) Improve Code Quality. Some of the aforementioned improvements could be identified automatically and realised by the help of program transformations. Similarly, our implementations of *library(dcg4pt)* and *library(plammar)* currently contain several predicates that are explicitly agnostic to their call mode – i. e., the execution order of goals stated in a rule's body only depends on which of its arguments are already bound. For instance, consider the definition of the predicate `prolog_tokens/3` in Listing 11.1. It is used to parse the Prolog source code given as a string in the first argument `String` and return the corresponding list of tokens, i. e., for the call mode `(+,?)`. Alternatively, a string should be returned in the first argument in case `Tokens` is already bound, i. e., for the call mode `(?,+)`. The order of the goals `I0` and `I1` varies because `string_chars/2`$_{\text{ISO}}$ throws a *not sufficiently instantiated* exception in case of two free variables, so in the second case, `I0` has to be called last.

This is because `string_chars/2`$_{\text{ISO}}$ describes a relationship between values. In *library(delay)*,[51] Michael Hendricks proposes to wrap these predicates in a `delay/1` predicate. Then, the variables are annotated via attributed variables (cf. Section 10.1.3) with their required call mode, and the actual predicate call happens as

[51] *library(delay)*, https://github.com/mndrix/delay, The Unlicense.

soon as it reasonably can. It is subject of future research to discuss how to automatically identify in a given Prolog program predicates that define constraints on their arguments.

Discussion on the Future of Prolog. In the long term, we see the potential for our *library(plammar)* to assist with the discussion of experimental new language features in Prolog. With its ability to easily define program transformations, *library(plammar)* could become what the open-source transpiler *Babel* currently is for the JavaScript community: a tool to experiment with new language constructs that are transpiled into plain old Prolog terms. This could significantly improve the dissemination of newly discussed flags, because they are realised only once in *library(plammar)* instead of concurrently and independently adding support in all major Prolog systems.

Bibliography

[1] Harvey Abramson. Definite Clause Translation Grammars. Technical report, 1984.

[2] Harvey Abramson and Verónica Dahl. *Logic Grammars (Symbolic Computation)*. Springer, 1989.

[3] Salvador Abreu and Daniel Diaz. Objective: In Minimum Context. In *Proc. 19th International Conference on Logic Programming (ICLP 2003)*, pages 128–147.

[4] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory*, volume 27. Springer Science & Business Media, 2002.

[5] Nicos Angelopoulos, Vítor Santos Costa, Joao Azevedo, Jan Wielemaker, Rui Camacho, and Lodewyk Wessels. Integrative Functional Statistics in Logic Programming. In *International Symposium on Practical Aspects of Declarative Languages (PADL 2013)*, pages 190–205. Springer, 2013.

[6] Dave Beckett and Brian McBride. RDF/XML Syntax Specification (Revised). *W3C Recommendation*, 10(2.3), 2004.

[7] Wouter Beek and Jan Wielemaker. SWISH: An Integrated Semantic Web Notebook. In *International Semantic Web Conference (Posters & Demos)*, 2016.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59:65–98, 2017.

[9] Stefan Bodenlos, Daniel Weidner, and Dietmar Seipel. PyPlC – Towards a Prolog Database Connectivity for Python. In *Proc. 32nd Workshop on Logic Programming (WLP 2018)*.

[10] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Pearson Education, 2001.

[11] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[12] Francisco Bueno, Daniel Cabeza, Manuel Carro, Manuel Hermenegildo, Pedro López-García, and Germán Puebla. *The Ciao Prolog System – Reference Manual*, 1997. System and online version of the manual available from https://ciao-lang.org.

[13] Hendrik Bünder. Decoupling Language and Editor – The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In *Proc. of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 129–140, 2019.

[14] Lawrence Byrd. Understanding the Control Flow of Prolog Programs. *Logic Programming Workshop*, 1980.

[15] Lee Byron. GraphQL: A Data Query Language, September 2015.

[16] Daniel Cabeza and Manuel Hermenegildo. A new Module System for Prolog. In *Proc. International Conference on Computational Logic (CL 2000)*, LNAI 1861, pages 131–148. Springer, 2000.

[17] Mats Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In *Proc. 4th International Conference on Logic Programming (ICLP 1987)*.

[18] Mats Carlsson and Thom Frühwirth. *SICStus Prolog User's Manual*. Books on Demand, 2014. Available from https://sicstus.sics.se/documentation.html.

[19] Weidong Chen. A Theory of Modules Based on Second-Order Logic. In *Proc. 4th IEEE International Symposium on Logic Programming*, pages 24–33, 1987.

[20] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer Science & Business Media, 2012.

[21] Alain Colmerauer. Les grammaires de métamorphose. Technical report, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy, 1975.

[22] Alain Colmerauer. Metamorphosis Grammars. In *Natural language communication with computers*, pages 133–188. Springer, 1978.

[23] Oracle Corp. MySQL 8.0 Reference Manual. 2020. Available from https://dev.mysql.com/doc/refman/8.0/en/.

[24] Vítor Santos Costa, Costa, Luis Damas, Rogério Reis, and Rúben Azevedo. *YAP User's Manual*, 2002. Available from https://www.dcc.fc.up.pt/~michel/yap.pdf.

[25] Michael A. Covington, Roberto Bagnara, Richard A. O'Keefe, Jan Wielemaker, and Simon Price. Coding Guidelines for Prolog. *Theory and Practice of Logic Programming*, 12(6):889–927, jun 2012.

[26] Michael A. Covington, Donald Nute, and André Vellino. *Prolog Programming in Depth*. Prentice Hall, 2nd edition, 1997.

[27] Douglas Crockford. JavaScript: The World's Most Misunderstood Programming Language. *Douglas Crockford's Javascript*, 2001.

[28] Jan C. Dageförde and Herbert Kuchen. A Constraint-Logic Object-Oriented Language. In *Proc. 33rd Annual ACM Symposium on Applied Computing (SAC 2018)*, pages 1185–1194, 2018.

[29] Jan C. Dageförde and Herbert Kuchen. A Compiler and Virtual Machine for Constraint-Logic Object-Oriented Programming with Muli. *Journal of Computer Languages (COLA)*, 53:63–78, 2019.

[30] Verónica Dahl and Michael C. McCord. Treating Coordination in Logic Grammars. *American Journal of Computational Linguistics*, 9(2):69–80, 1983.

[31] Bart Demoen. Dynamic Attributes, their hProlog Implementation, and a first Evaluation. *Report CW*, 350, 2002.

[32] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the Implementation of GNU Prolog. *Theory and Practice of Logic Programming*, 12(1-2):253–282, 2012.

[33] Mireille Ducassé and Jacques Noyé. Logic Programming Environments: Dynamic Program Analysis and Debugging. *The Journal of Logic Programming*, 19:351–384, 1994.

[34] Stéphane Ducasse. *Squeak: Learn Programming with Robots*. Apress, 2005.

[35] David C Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. *W3C Recommendation*, 16, 2004.

[36] Michael Fogus. *Functional JavaScript: Introducing Functional Programming with Underscore.js*. O'Reilly, 2013.

[37] Bryan Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[38] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[39] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock Roles, not Objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2004)*, pages 236–246, 2004.

[40] Steve Freeman and Nat Pryce. Evolving an Embedded Domain-Specific Language in Java. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006)*, pages 855–865, 2006.

[41] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[42] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for Knowledge Representation. In *Reasoning Web*, pages 104–124. Springer, 2008.

[43] Norbert E. Fuchs and Rolf Schwitter. Attempto Controlled English (ACE). In *Proc. 1st International Workshop on Controlled Language Applications (CLAW 1996)*.

[44] Shudi Gao, C Michael Sperberg-McQueen, Henry S Thompson, Noah Mendelsohn, David Beech, and Murray Maloney. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. *W3C Candidate Recommendation*, 2009.

[45] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proc. 22nd International Conference on Logic Programming (ICLP 2006)*, pages 41–55.

[46] Ken Hale. Endangered Languages: On Endangered Languages and the Safeguarding of Diversity. *Language*, 68:1–42, 1992.

[47] Olaf Hartig and Jorge Pérez. Semantics and Complexity of GraphQL. In *Proc. of the 2018 World Wide Web Conference*, pages 1155–1164, 2018.

[48] Günter Hegewald, Wilhelm Nickel, and Manfred Nogatz. Sortimentsoptimierung in der fleischverarbeitenden Industrie. *Aus der Arbeit des VEB Maschinelles Rechnen Dresden*, 7, 1970.
In memoriam of Manfred Nogatz, 1934–2019.

[49] Manuel Hermenegildo. A Documentation Generator for (C)LP Systems. In *Proc. International Conference on Computational Logic*, pages 1345–1361. Springer, 2000.

[50] Manuel Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F Morales, and Germán Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[51] Lynette Hirschman and Karl Puder. Restriction Grammar: A Prolog Implementation. *Logic Programming and its Applications*, pages 244–261, 1985.

[52] Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es), 1996.

[53] Paul Hudak. Modular Domain-Specific Languages and Tools. In *Proc. 5th International Conference on Software Reuse (ICSR 1998)*, pages 134–142, 1998.

[54] Patrick Ion and Robert Miner. Mathematical Markup Language (MathML) 1.0 Specification, July 1999. Available from https://www.w3.org/TR/REC-MathML/.

[55] ISO/IEC 13211-1. Information Technology – Programming Languages – Prolog – Part 1: General Core. ISO Standard, International Organization for Standardization, 1995.

[56] ISO/IEC 13211-2. Information Technology – Programming languages – Prolog – Part 2: Modules. ISO Standard, International Organization for Standardization, 2000.

[57] ISO/IEC 13211-2. Information Technology – Programming languages – Prolog – Part 3: Definite Clause Grammar Rules. ISO Standard, International Organization for Standardization, 2015.

[58] ISO/IEC 14977. Information Technology – Syntactic Metalanguage – Extended BNF. ISO Standard, International Organization for Standardization, 1996.

[59] Jona Kalkus. An Interactive Visualisation for Definite Clause Grammars. *Master Thesis, University of Würzburg, Germany*, 2017.

[60] Samuel N. Kamin and David Hyatt. A Special-Purpose Language for Picture-Drawing. In *Proc. USENIX Conference on Domain-Specific Languages (DSL 1997)*, pages 297–310. USENIX Association, 1997.

[61] Alan Kay. The Future of Programming as Seen from the 1960s. 2005. In the Foreword to [34].

[62] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. On the Implementation of the Probabilistic Logic Programming Language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.

[63] Donald E. Knuth. On the Translation of Languages from Left to Right. *Information and control*, 8(6):607–639, 1965.

[64] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[65] Tomaž Kosar, Pablo E. Martı, Pablo A. Barrientos, and Marjan Mernik. A Preliminary Study on Various Implementation Approaches of Domain-Specific Language. *Information and Software Technology*, 50(5):390–405, 2008.

[66] Robert Kowalski. Predicate Logic as Programming Language. In *IFIP Congress*, volume 74, pages 569–574, 1974.

[67] Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[68] Robert Kowalski. Time to Think like a Computer, 2011. Available from http://www.doc.ic.ac.uk/~rak/papers/NS.pdf.

[69] Julia Kübert. Attempto Controlled English für Amazon Alexa. *Bachelor Thesis, University of Würzburg, Germany*, 2018.

[70] Torbjörn Lager and Jan Wielemaker. Pengines: Web Logic Programming Made Easy. *Theory and Practice of Logic Programming*, 14(4-5):539–552, 2014. Demo Pengines application server available from https://github.com/SWI-Prolog/pengines.

[71] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS 1982)*, 4(2):258–282, 1982.

[72] Marjan Mernik. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2012.

[73] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[74] Dale Miller. A Logical Analysis of Modules in Logic Programming. *The Journal of Logic Programming*, 6:79–108, 1989.

[75] Alan Mycroft and Richard A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

[76] Falco Nogatz. CHR.js: Compiling Constraint Handling Rules to JavaScript. *Master Thesis, Ulm University, Germany*, 2015.

[77] Falco Nogatz and Thom Frühwirth. From XML Schema to JSON Schema: Translation with CHR. In *Proc. 11th International Workshop on Constraint Handling Rules*, 2014.

[78] Falco Nogatz, Thom Frühwirth, and Dietmar Seipel. CHR.js: A CHR Implementation in JavaScript. In *Rules and Reasoning (RuleML+RR 2018)*, volume 11092, pages 131–146. Springer, 2018.

[79] Falco Nogatz and Jona Kalkus. Declarative XML Schema Validation with SWI-Prolog. In Dietmar Seipel, Michael Hanus, and Salvador Abreu, editors, *Declarative Programming and Knowledge Management – Revised Selected Papers of Declare 2017*, pages 187–197, 2018.

[80] Falco Nogatz, Jona Kalkus, and Dietmar Seipel. Declarative XML Schema Validation with SWI-Prolog: System Description. In *Proc. 31st Workshop on (Constraint) Logic Programming (WLP 2017)*.

[81] Falco Nogatz, Jona Kalkus, and Dietmar Seipel. Web-based Visualisation for Definite Clause Grammars using Prolog Meta-Interpreters: System Description. In *20th International Symposium on Principles and Practice of Declarative Programming (PPDP 2018)*, pages 25:1–25:10. ACM, 2018.

[82] Falco Nogatz, Philipp Körner, and Sebastian Krings. Prolog Coding Guidelines: Status and Tool Support. In *Technical Communications of the 35th International Conference on Logic Programming (ICLP 2019)*.

[83] Falco Nogatz, Julia Kübert, Dietmar Seipel, and Salvador Abreu. Alexa, how can I reason with Prolog? In *8th Symposium on Languages, Applications, Technologies (SLATE 2019)*, volume 74 of *OpenAccess Series in Informatics (OASIcs)*, pages 17:1–17:9, 2019.

[84] Falco Nogatz and Dietmar Seipel. Implementing GraphQL as a Query Language for Deductive Databases in SWI-Prolog Using DCGs, Quasi Quotations, and Dicts. In *Proc. 30th Workshop on (Constraint) Logic Programming (WLP 2016)*.

[85] Falco Nogatz, Dietmar Seipel, and Salvador Abreu. Definite Clause Grammars with Parse Trees: Extension for Prolog. In *8th Symposium on Languages, Applications, Technologies (SLATE 2019)*, volume 74 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:14, 2019.

[86] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[87] Richard A. O'Keefe. An Elementary Prolog Library. Technical report, 2011. Available from http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm.

[88] George Souza Oliveira and Anderson Faustino da Silva. Towards an Efficient Prolog System by Code Introspection. In *Technical Communications of the 30th International Conference on Logic Programming (ICLP 2014)*.

[89] Ludwig Ostermayer. *Integration of Prolog and Java with the Connector Architecture CAPJa*. PhD thesis, Julius Maximilians University Würzburg, Germany, 2017.

[90] Ludwig Ostermayer, Frank Flederer, and Dietmar Seipel. CAPJA – A Connector Architecture for Prolog and Java. In *Proc. 10th Workshop on Knowledge Engineering and Software Engineering (KESE 2014)*.

[91] Ludwig Ostermayer, Frank Flederer, and Dietmar Seipel. PPI – A Portable Prolog Interface for Java. In *Proc. 28th Workshop on Logic Programming (WLP 2014)*, pages 38–52, 2014.

[92] John K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30, 1998.

[93] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) Parser Generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[94] Michael S. Paterson and Mark N. Wegman. Linear Unification. In *Proc. 8th annual ACM Symposium on Theory of Computing (STOC 1976)*, pages 181–186, 1976.

[95] Fernando Pereira. Extraposition Grammars. *American Journal of Computational Linguistics*, 7(4):243–256, 1981.

[96] Fernando Pereira and David 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.

[97] Fernando Pereira, David Warren, David Bowen, Lawrence Byrd, and Luis Pereira. C-Prolog User's Manual Version 1.5. Technical report, SRI International, 1988. Available from https://www2.cs.duke.edu/csl/docs/cprolog.html.

[98] John Peterson, Paul Hudak, and Conal Elliott. Lambda in Motion: Controlling Robots with Haskell. In *International Symposium on Practical Aspects of Declarative Languages (PADL 1999)*, pages 91–105. Springer, 1999.

[99] José A. Riaza. Tau Prolog: A Prolog Interpreter in JavaScript, 2017. Available from http://tau-prolog.org/.

[100] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[101] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a Language Server Protocol Infrastructure

for Graphical Modeling. In *Proc. of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 370–380, 2018.

[102] Christian Schneiker, Dietmar Seipel, Werner Wegstein, and Klaus Prätor. Declarative Parsing and Annotation of Electronic Dictionaries. In *6th International Workshop on Natural Language Processing and Cognitive Science (NLPCS 2009)*, pages 122–132, 2009.

[103] Dietmar Seipel. Knowledge Engineering for Hybrid Deductive Databases. In *Proc. 29th Workshop on Logic Programming (WLP 2015)*.

[104] Dietmar Seipel. Processing XML-Documents in Prolog. In *Proc. 17th Workshop on Logic Programming (WLP 2002)*.

[105] Dietmar Seipel. *PL4XML – An SWI-Prolog Library for XML Data Management (Manual)*, 2007. Available from http://www1.pub.informatik.uni-wuerzburg.de/databases/DisLog/fnq_manual.pdf.

[106] Dietmar Seipel and Joachim Baumeister. Declarative Specification and Interpretation of Rule-Based Systems. In *Proc. 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS 2008)*, pages 359–364. AAAI Press, 2008.

[107] Dietmar Seipel, Joachim Baumeister, and Marbod Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *Proc. 15th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2004)*, LNAI 3392, pages 16–31. Springer, 2005.

[108] Dietmar Seipel, Falco Nogatz, and Salvador Abreu. Prolog for Expert Knowledge Using Domain-Specific and Controlled Natural Languages. In *8th Language & Technology Conference: Human Language Technologies as a Challenge for Computer Science and Linguistics (LTC 2017)*, pages 138–140, 2017.

[109] Dietmar Seipel, Falco Nogatz, and Salvador Abreu. Domain-Specific Languages in Prolog for Declarative Expert Knowledge in Rules and Ontologies. *Computer Languages, Systems & Structures (COMLAN)*, 51C:102–117, 2018.

[110] Dietmar Seipel, Rüdiger von der Weth, Salvador Abreu, Falco Nogatz, and Alexander Werner. Declarative Rules for Annotated Expert Knowledge in Change Management. In *5th Symposium on Languages, Applications, Technologies (SLATE 2016)*.

[111] Paul Singleton, Fred Dushin, and Jan Wielemaker. JPL 3.0: A Bidirectional Prolog/Java Interface, 2004. Available from https://www.swi-prolog.org/packages/jpl and https://jpl7.org/.

[112] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming*, 29(1-3):17–64, 1996.

[113] J. Michael Spivey and Silvija Seres. Embedding Prolog in Haskell. In *Proceedings of Haskell*, volume 99, pages 1999–28, 1999.

[114] Leon Sterling and Ehud Y Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994.

[115] James M. Stichnoth and Thomas R Gross. Code Composition as an Implementation Language for Compilers. In *Proc. USENIX Conference on Domain-Specific Languages (DSL)*, pages 119–132. USENIX Association, 1997.

[116] Terrance Swift and David S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[117] Laurence Tratt. Domain-Specific Language Implementation via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS 2008)*, 30(6):1–40, 2008.

[118] Markus Triska. The Finite Domain Constraint Solver of SWI-Prolog. In *Proc. International Symposium on Functional and Logic Programming (FLOPS 2012)*, volume 7294 of *LNCS*, pages 307–316, 2012.

[119] Eric van der Vlist. *Relax NG: A Simpler Schema Language for XML*. O'Reilly, 2003.

[120] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[121] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Computer Science Division, UC Berkeley, 1990.

[122] Peter Van Roy. Extended DCG Notation: A Tool for Applicative Programming in Prolog. Technical Report UCB/CSD 90/583, Computer Science Division, UC Berkeley, 1990.

[123] Peter Van Roy, Bart Demoen, and Yves D Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. In *International Joint Conference on Theory and Practice of Software Development*, pages 111–125. Springer, 1987.

[124] Peter Van Roy and Alvin M. Despain. High-performance Logic Programming with the Aquarius Prolog Compiler. *Computer*, 25(1):54–68, 1992. Additional information about Aquarius Prolog available from https://www.info.ucl.ac.be/~pvr/aquarius.html.

[125] Didier Verna. Extensible Languages: Blurring the Distinction between DSL and GPL. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, pages 1–31. IGI Global, 2013. Published in [72].

[126] Rüdiger von der Weth, Dietmar Seipel, Falco Nogatz, Katrin Schubach, Alexander Werner, and Franz Wortha. Modellierung von Handlungswissen aus fragmentiertem und heterogenem Rohdatenmaterial durch inkrementelle Verfeinerung in einem Regelbanksystem. *Psychologie des Alltagshandelns*, 9(2):33–48, 2016.

[127] Jan Wielemaker. *SWI-Prolog Reference Manual 2.9.6*, 1990.

[128] Jan Wielemaker. SWI-Prolog: History and Focus for the Future. *ALP Issue*, 152, 2012.

[129] Jan Wielemaker. SWI-Prolog Version 7 Extensions. In *Proc. Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments (WLPE 2014)*, pages 109–123, 2014.

[130] Jan Wielemaker and Nicos Angelopoulos. Syntactic Integration of External Languages in Prolog. In *Proc. Workshop on Logic-based Methods in Programming Environments (WLPE 2012)*, pages 40–50, 2012.

[131] Jan Wielemaker and Anjo Anjewierden. PIDocPlDoc: Wiki style Literate Programming for Prolog. In *Proc. 17th Workshop on Logic-based Methods in Programming Environments (WLPE 2007)*.

[132] Jan Wielemaker, Wouter Beek, Michiel Hildebrand, and Jacco van Ossenbruggen. Cliopatria: A SWI-Prolog Infrastructure for the Semantic Web. *Semantic Web*, 7(5):529–541, 2016.

[133] Jan Wielemaker and Vítor Santos Costa. Portability of Prolog Programs: Theory and Case-Studies. *preprint arXiv:1009.3796*, 2010.

[134] Jan Wielemaker and Vítor Santos Costa. On the Portability of Prolog Applications. In *International Symposium on Practical Aspects of Declarative Languages (PADL 2011)*, pages 69–83. Springer, 2011.

[135] Jan Wielemaker, Leslie De Koninck, Thom Frühwirth, Markus Triska, and Marcus Uneson. *SWI-Prolog Reference Manual 7.1*. Books on Demand, 2014.

[136] Jan Wielemaker, Leslie De Koninck, Thom Frühwirth, Markus Triska, and Marcus Uneson. *SWI-Prolog Reference Manual*, 2021. Update of [135] for the latest stable SWI-Prolog version 8.4.1, November 2021. Available from https://www.swi-prolog.org/download/stable/doc/SWI-Prolog-8.4.1.pdf.

[137] Jan Wielemaker and Michael Hendricks. Why It's Nice to be Quoted: Quasiquoting for Prolog. In *Proc. 23rd Workshop on Logic-based Methods in Programming Environments (WLPE 2013)*.

[138] 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.

[139] Jan Wielemaker, Guus Schreiber, and Bob Wielinga. Prolog-Based Infrastructure for RDF: Scalability and Performance. In *International Semantic Web Conference*, pages 644–658. Springer, 2003.

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

[141] Allen Wirfs-Brock. ECMAScript Language Specification, ECMA-262, 6th Edition, June 2015. Available from https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[142] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017.

A

List of Contributions

I cannot begin to express to you how surreal this ride has been, because none of us grew up feeling like winners. So thank you to the bullies, to the popular kids, to the gym teachers who taunted us, who rejected us, and who made fun of the way we ran. Without you we never would have gone into comedy!

— Steve Levitan, Creator of "Modern Family", receiving the Emmy Award 2013

This appendix lists the author's contributions during the thesis that are related to the definition of and work with DSLs in Prolog, or logic-based programming languages in other host languages. It is a condensed version of Section 1.5.1, grouped by the work's type, with Appendix A.1 listing scientific contributions that have been published in journals, and those published in conference proceedings listed in Appendix A.2. The items are sorted chronologically, with the name of the author of this thesis each being highlighted.

For an exhaustive list and descriptions of assisting open-source software that was created by the author we refer to Section 1.5.2. Additional software that was developed during the thesis but is not directly related to the domain of discourse is listed in Appendix A.3.

A.1. Published in Journals

– Rüdiger von der Weth, Dietmar Seipel, **Falco Nogatz**, Katrin Schubach, Alexander Werner, and Franz Wortha. Modellierung von Handlungswissen aus fragmentiertem und heterogenem Rohdatenmaterial durch inkrementelle Verfeinerung in einem Regelbanksystem. *Psychologie des Alltagshandelns*, 9(2):33–48, 2016.

– Dietmar Seipel, **Falco Nogatz**, and Salvador Abreu. Domain-Specific Languages in Prolog for Declarative Expert Knowledge in Rules and Ontologies. *Computer Languages, Systems & Structures (COMLAN)*, 51C:102–117, 2018.

A.2. Published in Peer-Reviewed Conference Proceedings

- **Falco Nogatz** and Dietmar Seipel. Implementing GraphQL as a Query Language for Deductive Databases in SWI-Prolog Using DCGs, Quasi Quotations, and Dicts. In *Proc. 30th Workshop on (Constraint) Logic Programming (WLP 2016)*.

- Dietmar Seipel, Rüdiger von der Weth, Salvador Abreu, **Falco Nogatz**, and Alexander Werner. Declarative Rules for Annotated Expert Knowledge in Change Management. In *5th Symposium on Languages, Applications, Technologies (SLATE 2016)*.

- Dietmar Seipel, **Falco Nogatz**, and Salvador Abreu. Prolog for Expert Knowledge Using Domain-Specific and Controlled Natural Languages. In *8th Language & Technology Conference: Human Language Technologies as a Challenge for Computer Science and Linguistics (LTC 2017)*, pages 138–140, 2017.

- **Falco Nogatz**, Jona Kalkus, and Dietmar Seipel. Declarative XML Schema Validation with SWI-Prolog: System Description. In *Proc. 31st Workshop on (Constraint) Logic Programming (WLP 2017)*.

- **Falco Nogatz**, Thom Frühwirth, and Dietmar Seipel. CHR.js: A CHR Implementation in JavaScript. In *Rules and Reasoning (RuleML+RR 2018)*, volume 11092, pages 131–146. Springer, 2018.

- **Falco Nogatz** and Jona Kalkus. Declarative XML Schema Validation with SWI-Prolog. In Dietmar Seipel, Michael Hanus, and Salvador Abreu, editors, *Declarative Programming and Knowledge Management – Revised Selected Papers of Declare 2017*, pages 187–197, 2018.

- **Falco Nogatz**, Jona Kalkus, and Dietmar Seipel. Web-based Visualisation for Definite Clause Grammars using Prolog Meta-Interpreters: System Description. In *20th International Symposium on Principles and Practice of Declarative Programming (PPDP 2018)*, pages 25:1–25:10. ACM, 2018.

- **Falco Nogatz**, Dietmar Seipel, and Salvador Abreu. Definite Clause Grammars with Parse Trees: Extension for Prolog. In *8th Symposium on Languages, Applications, Technologies (SLATE 2019)*, volume 74 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:14, 2019.
  Honoured with the symposium's *Best Paper Award*.

- **Falco Nogatz**, Julia Kübert, Dietmar Seipel, and Salvador Abreu. Alexa, how can I reason with Prolog? In *8th Symposium on Languages, Applications,*

Technologies (SLATE 2019), volume 74 of *OpenAccess Series in Informatics (OASIcs)*, pages 17:1–17:9, 2019.

– **Falco Nogatz**, Philipp Körner, and Sebastian Krings. Prolog Coding Guidelines: Status and Tool Support. In *Technical Communications of the 35th International Conference on Logic Programming (ICLP 2019)*.

A.3. Additional Open-Source Software

In addition to the list of software that was created by the author as stated in Section 1.5.2, we developed the toolchain *tablediff* for efficient SQL table synchronisation:

– *tablediff*. https://github.com/fnogatz/tablediff, MIT License.
Shell scripts to get a minimal set of SQL commands for table synchronisation.
Toolchain that was developed for jfnetwork GmbH in Kitzingen, Germany, as part of a industry cooperation with the University of Würzburg.

B

Source Code Listings and Operator Tables

In this appendix we provide additional source code examples and definitions of operators which were shortened or split in the thesis before.

Contents

B.1. If-then Rules as an Internal Prolog DSL

In Section 5.4, if-then rules are defined as an internal Prolog domain-specific language by providing appropriate operator definitions by `op/3`[5.1]. It allows to specify expert knowledge in a natural-language-flavoured way, while still conforming to Prolog's syntax.

PROLOG

```
1  :- op(1100, yfx, then).
2  :- op(1000,  fx, if).
3  :- op( 900,  fx, [neg, not]).
4  :- op( 850, yfx, or).
5  :- op( 800, yfx, and).
6  %  op( 700, xfx, is). % is part of the ISO Prolog standard
7  :- op( 700, xfx, are).
8  :- op( 200,  fx, [a, an, the, no]).
9
10 % avoid name conflict with built-in is/2, cf. Section 5.2.3
11 :- redefine_system_predicate(is(_,_)).                    [5.2.3]
12
13 % if-then sentences can be stated as:
14 if the weather is rainy and there is no umbrella
15    or the weather is a thunderstorm
16 then the clothes are wet.
17 if the umbrella is broken then there is no umbrella.
18
19 % or as facts:
20 the weather is rainy.
21 the umbrella is broken.
```

Note the use of SWI-Prolog's `redefine_system_predicate/1`$_{\texttt{SWI}}$ directive (l. 11). It is required because the second if-then rule (l. 17) as well as the two facts (ll. 20–21) define clauses for the predicate `is/2`, which is built-in to calculate arithmetic expressions, see Section 3.2.1. For the sake of simplicity in this example, we do not provide a fallback to the built-in functionality of the system predicate `is/2`$_{\texttt{ISO}}$. Two techniques that avoid this breaking name conflict are introduced in Section 5.2.3.

B.2. EBNF Grammar Rules as an Internal Prolog DSL

In Section 5.6, we discuss how to define and use the extended Backus–Naur form (EBNF) as an internal domain-specific language in Prolog. With appropriate operators for the symbols defined for EBNF in [58], only minor modifications are required to embed a given EBNF verbatim into Prolog source code.

```
1  :- op(1150, xfy, ;).                                                    Prolog
2  :- op(1100, xfx, =).
3  :- op(1001, xfy, '|').
4  %  op(1000, xfy, ',').  % is part of the ISO Prolog standard
5
6  % avoid name conflict with built-in ;/2, cf. Section 5.2.3
7  :- redefine_system_predicate(;(_,_)).                                   [5.2.3]
8                                                                          Ebnf Dsl
9                variable = [ layout_text_sequence /* 6.4.1 */ ],
10                          variable_token /* 6.4.3 */ ;
11         variable_token = anonymous_variable /* 6.4.3 */
12                        | named_variable /* 6.4.3 */ ;
13     anonymous_variable = variable_indicator_char /* 6.4.3 */ ;
14         named_variable = variable_indicator_char /* 6.4.3 */,
15                          alphanumeric_char /* 6.5.2 */,
16                          { alphanumeric_char /* 6.5.2 */ }
17                        | capital_letter_char /* 6.5.2 */,
18                          { alphanumeric_char /* 6.5.2 */ } ;
19 variable_indicator_char = underscore_char /* 6.5.2 */ ;
20        underscore_char = "_" ;
21    capital_letter_char = "A" | "B" | "C" | ... ;
22      alphanumeric_char = ... . % sic!
```

With the operator definitions of lines 1–4, the EBNF given in lines 9–22 is a valid Prolog clause, with a nested compound term of `;/2` as its principal functor. The types of the operators `;/2`, `=/2`, and `'|'/2` are as defined by the ISO Prolog standard. Their precedences are restricted by the values for $\texttt{','/2}_{\texttt{ISO}}$ and `'|'/2`: if the comma is defined as an operator, it has to be of precedence 1000 (Iso 6.3.4.3). The bar operator `'|'/2` on the other hand is required to be only an infix operator, and with precedence greater than or equal to 1001. The terms that are enclosed in square brackets (l. 9), and curly brackets (ll. 16, 18) are valid without further operator definitions, as they constitute a curly bracketed term and compound terms in list notation following the ISO Prolog standard (Iso 6.3).

If integrated into an existing Prolog source code file, the presented code fragment should be stated as the very last, since the overwriting of Prolog's built-in operator precedences for `;/2`$_{\text{ISO}}$ and `=/2`$_{\text{ISO}}$ will likely result in incompatibilities. This can be avoided by using a separate module for code of the internal DSL, because in SWI-Prolog, operators are local to a module.

Note that `...` is a valid atom in Prolog, so the presented listing can be read in as a Prolog program without any modifications, even with the ellipses of lines 21 and 22. The EBNF of lines 9–22 then reads as the following Prolog program written in canonical notation with the classical operator `./2` for the list construction:

```
;( =( variable, ’,’(.( layout_text_sequence, [] ), variable_token )),
   ;( =( variable_token, ’|’( anonymous_variable, named_variable )),
      ;( =( anonymous_variable, variable_indicator_char ), ;(/*...*/)
   ))).
```

A longer form of this term in canonical notation is given at the end of Section 5.6.

B.3. Expanding EBNF Grammar Rules to DCGs

An EBNF that is given in the form of an internal Prolog DSL as shown in Appendix B.2 can be translated into plain old definite clause grammars via the following term expansion. This technique is used in Section 9.2 to directly use the Prolog syntax specification given in the ISO Prolog standard, resulting in an executable Prolog program that parses and serialises Prolog code. There, we additionally make use of *library(dcg4pt)* to also process the corresponding parse tree on the fly. The following listing instead focusses only on the translation of EBNF to plain old DCGs, without further arguments.

```
1  :- op(800, fx, *).                                          PROLOG
2  :- op(800, fx, ?).
3  ebnf_rule(X1 = Y1, X1 --> Y2) :- ebnf_body(Y1, Y2).
4  ebnf_body(T1, T2) :-
5    T1 =.. [F, X1, Y1],                                       [3.6]
6    memberchk(F, [(,), ’|’]),                                 [C.10]
7    ebnf_body(X1, X2),
8    ebnf_body(Y1, Y2),
9    T2 =.. [F, X2, Y2].                                       [3.6]
10 ebnf_body({ X1 }, *X1).
11 ebnf_body([ X1 ], ?X1).
12 ebnf_body(X, X).
```

```
13
14 term_expansion(X1, X2s) :-
15   X1 = (_ ; _),
16   term_functors_list(X1, [(;)], X1s),                    [C.3]
17   maplist(ebnf_rule, X1s, X2s).                          [C.11]
```

The term expansion (ll. 14–17) applies only for terms with a principal functor of `;/2`. This precondition ensures it is used only for our internal DSL. The various EBNF production rules are split by the infix operator `;/2` into the list `X1s`, and each is translated separately using the predicate `ebnf_rule/2`. Consequently, the single compound term `X1` is expanded to the list of DCGs that is returned in `X2s`. The created DCGs are then transformed to Prolog predicates via SWI-Prolog's `dcg_translate_rule/2`$_{\texttt{SWI}}$. Only after that, SWI-Prolog's compilation reaches a fixpoint (cf. Section 5.2.1), with no terms that can be further expanded.

The prefix operators `*/1` and `?/1` are defined in lines 1–2, and are used to represent EBNF's (possibly empty) repetitions and optionals (ll. 10–11). Because of the term expansion to Prolog clauses with difference lists, they describe the predicates `*/3` and `?/3` – their first argument is the DCG body that is optional or repeated, the second and third represent the list and its remainder.

As emphasised before, the code of Appendix B.2 is required to be stated only after the aforementioned term expansion, as otherwise the internal DSL's differing operator precedences for `;/2`$_{\texttt{ISO}}$ and `=/2`$_{\texttt{ISO}}$ break the definition of the referenced predicates like `memberchk/2`$_{[C.8]}$.

B.4. Meta-Interpreter for DCGs

In Section 6.2.3, the built-in meta-predicate `phrase/3`$_{\texttt{ISO}}$ is defined in the form of a meta-interpreter for definite clause grammars. Since DCGs are usually translated into plain old Prolog clauses via term expansion, we use the infix operator `--->/2` instead to hold grammar rules. Otherwise, Prolog's expansion for `-->/2`$_{\texttt{ISO}}$ has to be deactivated to be able to use the original grammar rules in a meta-interpreter.

```
1 :- op(1200, xfx, --->). % instead of -->/2                    PROLOG
2 % DCG as in Listing 6.1
3 elem(X)    ---> [X], { member(X, [a,b,c]) }.
4 palindrome ---> [].
5 palindrome ---> elem(_).
6 palindrome ---> elem(X), palindrome, elem(X).
```

```
7
8  % declare built-in predidcates phrase/{2,3} to be redefined
9  :- redefine_system_predicate(phrase(_,_)).                    [5.2.3]
10 :- redefine_system_predicate(phrase(_,_,_)).                  [5.2.3]
11
12 %% phrase(:Body, ?List)
13 phrase(Body, List) :- phrase(Body, List, []).
14
15 %% phrase(:Body, ?List, ?Rest)
16 % nonterminals
17 phrase(Head, A, Z) :- (Head ---> Body), phrase(Body, A, Z).
18 phrase(Head, A, Z) :-
19   (Head, Pushback ---> Body),
20   phrase(Body, A, B),
21   append(B, Pushback, Z).                                      [3.4]
22 phrase((B1 , B2), A, Z) :- phrase(B1, A, D) , phrase(B2, D, Z).
23 phrase((B1 ; B2), A, Z) :- phrase(B1, A, Z) ; phrase(B2, A, Z).
24 phrase(\+ H, A, A) :- \+ phrase(H, A, _).                    [3.3]
25 phrase({ P }, A, A) :- call(P).
26
27 % terminals
28 phrase([], A, A).
29 phrase([T|Rest], [T|A], Z) :- phrase(Rest, A, Z).
```

B.5. Term Expansion for DCGs

In Section 6.2.4, the idea of translating DCGs into plain old Prolog clauses via term expansions is introduced. As before in Appendix B.4, we use the operator `--->/2` instead of `-->/2`$_{\text{ISO}}$ to denote DCGs while avoiding conflicts with Prolog's built-in term expansions.

```
1  :- op(1200, xfx, --->).                                       PROLOG
2  term_expansion(X1 ---> Y1, X2 :- Y2) :-
3    ( X1 = (L, P), append(P, Z, Out)                            [3.4]
4    ; X1 = L, Out = Z),
5    term_args_attached(L, [In, Out], X2),                       [C.4]
6    translate_body(Y1, Y2, In, Z).
7
8  %% translate_body(+,-,?,?)
```

```
9  % terminals
10 translate_body(L, true, A, Z) :-
11   is_list(L),                                                    [3.3.1]
12   append(L, Z, A).                                               [3.4]
13 % nonterminals
14 translate_body((B1 , B2), (C1 , C2), A, Z) :-
15   translate_body(B1, C1, A, D),
16   translate_body(B2, C2, D, Z).
17 translate_body((B1 ; B2), (C1 ; C2), A, Z) :-
18   translate_body(B1, C1, A, Z),
19   translate_body(B2, C2, A, Z).
20 translate_body(\+ H1, \+ H2, A, A) :- translate_body(H1, H2, A, _).
21 translate_body(!, !, A, A).
22 translate_body({ P }, P, A, A).
23 translate_body(Nonterminal, Predicate, A, Z) :-
24   term_args_attached(Nonterminal, [A, Z], Predicate).            [C.4]
25
26 % DCG as in Listing 6.1
27 elem(X)    ---> [X], { member(X, [a,b,c]) }.
28 palindrome ---> [].
29 palindrome ---> elem(_).
30 palindrome ---> elem(X), palindrome, elem(X).
```

In contrast to the handling of DCGs in the meta-interpreter, we split the expansion into two parts. In lines 2–6, a grammar rule with its head gets translated. For translation of the rule's right-hand side, it relies on the predicate `translate_body/4`, which is defined in lines 8–24. It takes a DCG body as its first argument, returns the created Prolog goal in the second argument, and holds two additional arguments that represent the processed difference list.

The different cases for the DCG's right-hand side can be adapted from the meta-interpreter. For each body element, two arguments representing the difference list have to be added, and new chaining variables have to be introduced accordingly. We changed the order of terminals and nonterminals to move the most general case of a nonterminal to the very ending, as otherwise it would require additional type checks – e. g., `List1 \= ['.'|_], List1 \= [{}|_]` to avoid `Nonterminal` to be a list or curly bracketed term again.

B.6. Meta-Interpreter for DCGs with Tracing

In Section 7.3.3, we introduce a modification for the meta-interpreter for DCG to support tracing of backtracking and failing subgoals. There, the definition of the used predicate `print_indented/2` that prints a term with a given indentation level is given. As before, we use the operator `--->/2` instead of `-->/2`$_{\text{ISO}}$ to denote DCGs in order to avoid conflicts with Prolog's built-in term expansions.

PROLOG

```
%% phrase(:Body, +Level, ?List, ?Rest)
% nonterminals
phrase(Head, L, A, Z) :-
  LL is L+1,
  ( (Head ---> Body),
    print_indented((Head ---> Body):call, L),                    [7.3.3]
    phrase(Body, LL, A, Z)
  ; print_indented(Head:fail, L)).                               [7.3.3]
phrase(Head, L, A, Z) :-
  LL is L+1,
  ( (Head, Pushback ---> Body),
    print_indented((Head, Pushback ---> Body):call, L),          [7.3.3]
    phrase(Body, LL, A, C),
    append(C, Pushback, Z)                                       [3.4]
  ; print_indented(Head:fail, L)).                               [7.3.3]
phrase((B1 , B2), L, A, Z) :-
  phrase(B1, L, A, D),
  phrase(B2, L, D, Z).
phrase((B1 ; B2), L, A, Z) :-
  phrase(B1, L, A, Z);
  phrase(B2, L, A, Z).
phrase(\+ H, L, A, A) :-
  ( \+ phrase(H, A, _),
    print_indented((\+ phrase(H, A, _)):exit, L)                 [7.3.3]
  ; print_indented((\+ phrase(H, A, _)):fail, L)).               [7.3.3]
phrase({ P }, L, A, A) :-
  ( call(P),                                                     [3.6]
    print_indented(call(P):exit, L)                              [7.3.3]
  ; print_indented(call(P):fail, L)).                            [7.3.3]
phrase(!, L, A, A) :- !, print_indented(cut, L).                 [7.3.3]
% terminals
phrase([], L, A, A) :- print_indented(empty, L).                 [7.3.3]
```

```
33 phrase([T|Rest], L, [T|A], Z) :-
34   print_indented(T:consume, L),                                    [7.3.3]
35   phrase(Rest, L, A, Z).
```

The meta-interpreter expects an additional argument `L` to hold the current level of used nonterminals. It is incremented only in case of a newly consumed grammar rule (ll. 3–15), and can be used, e. g., as a recursion limit.

B.7. Meta-Nonterminal `sequence//3`

In Section 8.3, we present the modified term expansion scheme for DCGs with automatic parse tree processing. For sequences of nonterminals, we provide the meta-nonterminal `sequence//3`, with the following meaning: `sequence(+Mode, :`*NT*`(Arg`$_1$`,...,Arg`$_n$`), ?PTs)` is a sequence of the nonterminal *NT*`//`*n* with arguments `Arg`$_i$, the sequence's corresponding list `PTs` of parse trees, and the repetition mode `Mode` being one of the atoms `?`, `*`, `**`, and `+`. A more detailed description of the nonterminal `sequence//3` and its corresponding Prolog predicate `sequence/5` is given in Section 8.3, together with a short description of the supported modes.

```
1 %% sequence(?Mode, :DCGBody, ?ParseTrees, ?A, ?Z)                  Prolog
2 :- meta_predicate sequence(?, //, ?, ?, ?).                        [3.6]
3 sequence( ?, B, [P]) --> call(B, P).                               [3.6]
4 sequence( ?, _, []) --> [].
5 sequence( *, _, []) --> [].
6 sequence( *, B, [P|Ps]) --> call(B, P), sequence( *, B, Ps).       [3.6]
7 sequence(**, B, [P|Ps]) --> call(B, P), sequence(**, B, Ps).       [3.6]
8 sequence(**, _, []) --> [].
9 sequence( +, B, [P|Ps]) --> call(B, P), sequence( *, B, Ps).       [3.6]
```

With the standard term expansion for DCGs from Section 6.2.4, the nonterminal `sequence//3` is translated into the Prolog predicate `sequence/5` at compile-time. Similarly, each subgoal `?- call(DCGBody, PT)` in the grammar rule's right-hand side is automatically extended by the two additional arguments `A` and `Z` that hold the processed difference list. This way, the body of the resulting Prolog clause after the term expansion of the DCG from line 3 becomes `?- call(DCGBody, PT, A, Z)`, using the built-in predicate `call/4`$_{\text{ISO}}$.

B.8. Transformation for DCG Bodies with Parse Trees

In Section 8.3, we describe the source-to-source transformation for DCGs to additionally automatically process the corresponding parse tree on execution. Similar to the standard term expansion scheme for DCGs, this transformation is split into two parts: the translation of the grammar rule's left-hand side, and the body elements on the other. The latter is defined in the predicate `dcg4pt_formula_to_dcg_formula/3`, for which we presented an extract in Listing 8.5.

```
%% dcg4pt_formula_to_dcg_formula(+Y1, -Y2, ?Value)                    PROLOG
dcg4pt_formula_to_dcg_formula([T], [T], T).          % terminals
dcg4pt_formula_to_dcg_formula(Ts, Ts, Ts) :- is_list(Ts).          [3.3.1]
dcg4pt_formula_to_dcg_formula(Y1, Y1, Y1) :- string(Y1).           [3.3.4]

dcg4pt_formula_to_dcg_formula(Y1, Y2, V) :-          % conjunction
  Y1 = (_, _),
  term_functors_list(Y1, [(,)], Ys1),                                [C.3]
  maplist(conj_body, Ys1, Ys2, R0s, R1s),                           [C.11]
  R0s = [V|R0s_],
  append(R1s_, [Last], R1s),                                         [3.4]
  Last = [],
  maplist((=), R0s_, R1s_),                                         [C.11]
  term_functors_list(Y2, [(,)], Ys2).                                [C.3]

dcg4pt_formula_to_dcg_formula(Y1, Y2, V) :-          % disjunction
  (Y1 = (_ ; _) ; Y1 = (_ | _)),
  term_functors_list(Y1, [(;), '|'], Ys1),                           [C.3]
  maplist(dcg4pt_formula_to_dcg_formula, Ys1, Ys2, Vs),             [C.11]
  maplist(add_variable_binding(V), Ys2, Vs, Ysn2),                  [C.11]
  term_functors_list(Y2, [(;)], Ysn2).                               [C.3]

dcg4pt_formula_to_dcg_formula(!, !, []).             % cut
dcg4pt_formula_to_dcg_formula({ P }, { P }, []).     % embedded Prolog
dcg4pt_formula_to_dcg_formula(\+ Y1, \+ Y2, _) :-    % negation
  dcg4pt_formula_to_dcg_formula(Y1, Y2, _).
dcg4pt_formula_to_dcg_formula(X1, X2, V) :-          % nonterminal
  term_args_attached(X1, [V], X2).                                   [C.4]
```

The code for terminals and conjunctions is described together with the predicate's underlying principles in Section 8.3. For conjunctions in a grammar rule's right-hand side, each element is transformed using the auxiliary predicate `conj_body/4`:

```
29 %% conj_body(+Y1, -Y2, ?List, ?Remainder)                    PROLOG
30 conj_body(A, B, R0, R1) :-
31   A = *(C), !,
32   conj_body(sequence('*', C), B, R0, R1).
33 conj_body(A, B, R0, R1) :-
34   A = ?(C), !,
35   conj_body(sequence('?', C), B, R0, R1).
36 conj_body(A, B, R0, R1) :-
37   A = sequence(_, _), !,
38   dcg4pt_formula_to_dcg_formula(A, DCGBody, V),
39   B = call_sequence_ground(DCGBody, V, R1, R0).              [8.4.3]
40 conj_body(A, B, R0, R1) :-
41   dcg4pt_formula_to_dcg_formula(A, DCGBody, V),
42   B = ({ R0 = [V|R1] }, DCGBody).
```

While the conjunction (ll. 6–14) uses chaining variables to compose the overall inner value `V`, each body element serves as an alternative in case of the disjunction via `;/2` or `|/2`. Therefore, all modified body elements in `Ys2` bind to the same value `V` (ll. 19–20), which is done with the help of `maplist/4`[C.11] and `add_variable_binding/4`, which is defined as follows:

```
43 %% add_variable_binding(?To, +Y1, ?Value, -Y2)               PROLOG
44 add_variable_binding(V, Y1, Bind, ({ Bind = V }, Y1)).
```

In the definition of `dcg4pt_formula_to_dcg_formula/3`, the case for nonterminals is specified as the very last, as it allows to omit the check for a callable term via `callable/1`$_{\text{ISO}}$. Our implementation also does not rely on the cut `!/0`$_{\text{ISO}}$, as SWI-Prolog does not backtrack over term expansions in its compilation process, as described in Section 5.2.

B.9. Meta-Predicates `*/4` and `?/4`

In Section 9.2.1, we describe the transformation process from an EBNF given in the form of an internal Prolog DSL to an equivalent DCG with an additional argument for an options list and the corresponding parse tree. We define short notations for optionals and repetitions in DCGs in the second transformation step, as they are

frequently used in the ISO Prolog standard. The used prefix operators `?/1` and `*/1` become predicates of arity 4 in the subsequent program transformation steps. Following our considerations of Section 8.4 to support both efficient parsing and serialisation, the repetitions are mapped to sequences of either mode `**` or `*`, depending on whether the processed difference list is bound, or the parse tree.

```
%% *(:DCGBody, ?ParseTrees, ?A, ?Z)                                    PROLOG
*(DCGBody, PTs, A, Z) :-
  nonvar(A), !, % use `**` to consume as most as possible              [3.2.1]
  sequence(**, DCGBody, PTs, A, Z).                                    [B.7]
*(DCGBody, PTs, A, Z) :-
  var(A), !, % use `*` to produce as small as possible                 [3.2.1]
  sequence(*, DCGBody, PTs, A, Z).                                     [B.7]
?(DCGBody, PTs, A, Z) :-
  sequence(?, DCGBody, PTs, A, Z).                                     [B.7]
```

B.10. Operator Types and Their Precedence Constraints

In Section 10.1.4, we introduce SWI-Prolog's *library(clpfd)*, which allows to define arithmetic relations about integer variables of finite domains. With the help of the constraints `#</2`$_{\texttt{SWI}}$ and `#=</2`$_{\texttt{SWI}}$, our *library(plammar)* defines the predicates `prec_constraints/`$\{3,4\}$, which realise the dependencies of the various operator precedences that are implicitly given by their types.

```
%% prec_constraints(+Type, ?Op, ?Term1, ?Term2)                        PROLOG
%% prec_constraints(+Type, ?Op, ?Term)
prec_constraints(xfx, P_Op, P_Term1, P_Term2) :-
  P_Term1 #<  P_Op, P_Term2 #<  P_Op.                                  [10.1.4]
prec_constraints(yfx, P_Op, P_Term1, P_Term2) :-
  P_Term1 #=< P_Op, P_Term2 #<  P_Op.                                  [10.1.4]
prec_constraints(xfy, P_Op, P_Term1, P_Term2) :-
  P_Term1 #<  P_Op, P_Term2 #=< P_Op.                                  [10.1.4]
prec_constraints(xf,  P_Op, P_Term) :- P_Term #<  P_Op.                [10.1.4]
prec_constraints(yf,  P_Op, P_Term) :- P_Term #=< P_Op.                [10.1.4]
prec_constraints( fx, P_Op, P_Term) :- P_Term #<  P_Op.                [10.1.4]
prec_constraints( fy, P_Op, P_Term) :- P_Term #=< P_Op.                [10.1.4]
```

B.11. Operator Inference for If-then Rules

In Section 5.4, we define an internal Prolog DSL to express knowledge in the form of if-then rules, so that the example sentence "if the weather is rainy and there is no umbrella or the weather is a thunderstorm then the clothes are wet" from Listing 5.6 is a valid Prolog program. With *library(plammar)*, the required operator definitions can be inferred from this example sentence. In the following, we define the predicate `solution/1`, which returns a list of all required operator definitions. A possible computed answer substitution is given in Listing 10.7.

```
:- use_module(library(plammar)).                                    PROLOG
solution(Ops) :-
  Ops = [
    op(700, xfx, is), op(700, xfx, are),
    op(_, fx, a), op(_, fx, the), op(_, fx, no) | _],
  NotOps = [
    op(_, _, weather), op(_, _, rainy),
    op(_, _, umbrella), op(_, _, thunderstorm),
    op(_, _, clothes), op(_, _, wet) ],
  Input = string("if the weather is rainy and there is no umbrella or
     the weather is a thunderstorm then the clothes are wet."),
  prolog_parsetree(Input, _PT,
    [infer_operators(Ops), disallow_operators(NotOps)]).
```

From the example, it is known that words like `weather` and `rainy` are entities the if-then rules are about. Thus they should not be operators, which is why they are passed to `prolog_parsetree/3` in its option `disallow_operators`. The words `is` and `are` should be defined as operators in the same way $\texttt{is/2}_{\texttt{ISO}}$ is defined in the ISO Prolog standard; determiners on the other hand are known to be operators of type `fx`.

B.12. XML Schema 1.1 with Embedded XPath Expressions

In Section 10.4, we define XPath expressions as an internal Prolog DSL. It is used by our *library(xsd)* to support features that were introduced in version 1.1 of the XML Schema Definition [44], namely assertions and conditional type expressions. In the following, we present an exemplary XML Schema that relies on all of these features.

XML Schema

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:fn="http://www.w3.org/2005/xpath-functions">
  <xs:element name="list" type="list">
    <xs:key name="entry-id-key">
      <xs:selector xpath="entry" />
      <xs:field xpath="@id" />
    </xs:key>
  </xs:element>
  <xs:complexType name="list">
    <xs:sequence>
      <xs:element name="entry" maxOccurs="unbounded" type="entry">
        <xs:alternative test="@class = 'even'" type="entry-even" />
      </xs:element>
    </xs:sequence>
    <xs:assert test="entry[@id mod 2 eq 0]/@class = 'even' or
                     entry[@id mod 2 eq 1]/@class = 'odd'" />
    <xs:assert test="every $i in 1 to count(entry)-1 satisfies
                       (entry[$i] lt entry[$i+1])" />
  </xs:complexType>
  <xs:complexType name="entry">
    <xs:simpleContent>
      <xs:extension base="day-24">
        <xs:attribute name="id" type="xs:positiveInteger" />
        <xs:attribute name="class" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="entry-even">
    <xs:complexContent>
      <xs:extension base="entry">
        <xs:attribute name="bgColor" type="xs:string" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:simpleType name="day-24">
    <xs:restriction base="xs:date">
      <xs:assertion test="fn:day-from-date($value) eq 24" />
    </xs:restriction>
```

```
  </xs:simpleType>
</xs:schema>
```

An example XML document that is valid according to this schema is:

```XML
<list>
  <entry id="1" class="odd">1990-01-24</entry>
  <entry id="2" class="even" bgColor="red">1993-05-24</entry>
  <entry id="3" class="odd">2021-12-24</entry>
</list>
```

B.13. Operators for XPath as an Internal Prolog DSL

As discussed in Section 10.4, the following user-defined operators allow to treat XPath as an internal Prolog DSL:

Precedence	Type	Operators
950	xfx	satisfies then
900	xfx	else
900	fx	every some if
850	yfx	or
800	yfx	and
750	xf	![1]
700	xfx	*= ~= eq ne le lt ge gt in[2] >[3] <[3]
400	yfx	~
400	fy	/ /@ ./ ./@ // //@ .// .//@
400	yfx	/[3] /@ // //@ :: ::* mod[3] idiv
400	xfx	to
400	yf	[][4]
200	fy	@
1	fx	$[5]

[1] For usage in `!=`, which does not constitute a single token.
[2] Following the definition in *library(clpfd)*.
[3] Built-in operator defined in the ISO Prolog standard.
[4] Requires Prolog language extension `allow_square_block_op`[D.17].
[5] Built-in operator in SWI-Prolog [136, Sec. 4.40].

In our *library(xsd)*, these operator definitions are specified in the sub-module *library(xsd/xpath)*, which can be loaded via `?- use_module(library(xsd/xpath))`.

C

Non-Standard Definition of Predicates and Operators

In this appendix we provide the definitions of proprietary Prolog predicates that we use and refer to in this thesis but which are either not part of the ISO Prolog standard, not provided by all major Prolog implementations, or implemented differently. Where indicated, the predicates are included in SWI-Prolog or can be automatically loaded via additional libraries. The definitions given here differ from SWI-Prolog's implementation, as we do not rely on additional user-defined predicates.

Contents

C.1. Predicates `call/`n

```
1 %% call(:Closure, ExtraArg1)                                    PROLOG
2 :- meta_predicate call(1, ?).                                    [3.6]
3 call(Closure, ExtraArg1) :-
4   Closure =.. [P|Args],                                          [3.6]
5   append(Args, [ExtraArg1], AllArgs),                            [3.4]
6   Goal =.. [P|AllArgs],                                          [3.6]
7   Goal.                                                          [3.6]
```

The meta-predicate `call/2`$_{\texttt{ISO}}$ is used to dynamically create and call a goal. The goal `?- call(Closure, ExtraArg1)` appends `ExtraArg1` to the argument list of the compound term `Closure` and calls the result.

In SWI-Prolog, all calls for predicates of the `call/`$n_{\texttt{ISO}}$ family with $n \geq 2$ are directly handled by the compiler [136, Sec. 4.8]. Only to allow reflection and code listings, the predicates of arity $2 \leq n \leq 8$ are also implemented as plain old Prolog predicates. Our implementation can be adapted to any predicate `call/`n of arity n by adding further variables `ExtraArg`$_i$ to the clause's head (l. 3), and to the call for `append/3`[3.4] (l. 5).

C.2. Predicate and Operator `\+/1`

```
1 %% \+(:Goal)                                                    PROLOG
2 :- op(900, fy, \+).                                              [5.1]
3 \+ Goal :-
4   call(Goal),                                                    [3.6]
5   !,                                                             [3.2.2]
6   false.                                                         [ISO]
7 \+ Goal.
```

The predicate `\+/1`$_{\texttt{ISO}}$ is used to indicate a negated goal under the closed-world assumption, i. e., a goal that is not provable (cf. Section 3.2). It can be implemented with the help of the cut `!/0`$_{\texttt{ISO}}$. If the goal `Goal` terminates existentially with a solution, the goal `\+(Goal)` is known to be false (ll. 3–6). Only if there is no solution for `Goal`, `\+(Goal)` is true (l. 7). The cut is required here to prevent the undesired backtracking after the explicit `false/0`$_{\texttt{ISO}}$. The predicate `\+/1`$_{\texttt{ISO}}$ is often used in the form of a unary prefix operator (l. 2).

C.3. Predicate `term_functors_list/3`

```
%% term_functors_list(+Term, +Functors, -List)                    PROLOG
%% term_functors_list(-Term, +Functors, +List)
term_functors_list(Term, Names, [A,B|Rest]) :-
  member(Name, Names),                                            [SWI]
  Term =.. [Name, A, TermB],                                      [3.6]
  term_functors_list(TermB, Names, [B|Rest]).
term_functors_list(A, _, [A]).
```

The predicate `term_functor_list/3` can be used to compose (instantiation mode `(-,+,+)`) and decompose (mode `(+,+,-)`) a compound term from the name of a binary operator and the list of operands. It works for both left-associative and right-associative operators. In `Functors`, a list of possible operators is expected. For instance, by calling `?- term_functors_list(Term, [(;), '|'], [a, b])`, the two alternative notations for disjunctions, `Term = (a ; b)` and `Term = (a | b)`, are generated.

C.4. Predicate `term_args_attached/3`

```
%% term_args_attached(+Term1, +List, -Term2)                      PROLOG
%% term_args_attached(-Term1, +List, +Term2)
%% term_args_attached(+Term1, -List, +Term2)
term_args_attached(Term1, List, Term2) :-
  Term1 =.. U1,                                                   [3.6]
  append(U1, List, U2),                                           [3.4]
  Term2 =.. U2.                                                   [3.6]
```

When working with compound terms, it is a common task to create a new term by amending an existing by new arguments. One use case is the insertion of the two arguments that represent the processed difference list when translating a DCG into plain old Prolog clauses (cf. Section 6.2.4 and Appendix B.5); another for the adding of the parse tree argument in the source-to-source transformation of DCGs (cf. Section 8.3). The goal `?- term_args_attached(Term1, List, Term2)` is true if `Term2` is the compound term `Term1` with the additional arguments given in the list `List`. The predicate is part of our *library(dcg4pt)* and supports three instantiation modes. Given two bound arguments, the missing part gets calculated.

C.5. Predicate `char_code/2`

```
1  :- set_prolog_flag(double_quotes, chars).                    PROLOG
2
3  %% char_code(?Character, ?Code)
4  char_code(Char, Code) :-
5    Alphabet = " !\"#$%&'()*+,-./0123456789:;<=>?@\
6                ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`\
7                abcdefghijklmnopqrstuvwxyz{|}~",
8    append(Before, [Char|_], Alphabet),                          [3.4]
9    length(Before, Position),                                    [ISO]
10   Code is Position+32.                                         [3.2.1]
```

The predicate `char_code/2`$_{\text{ISO}}$ can be used to convert between a single character (i. e., an atom of length 1) and its character code. It is part of the ISO Prolog standard. Traditionally, Prolog is based on a 7-bit US-ASCII character set, i. e., the code is between 0 and 127. Our implementation lists only the printable characters, beginning with the space character, which has a code of 32. We specify the list of possible characters as a double quoted list (ll. 6–8). Because of the program flag in line 1 (cf. Section 3.3.4), it evaluates to a list of atoms, and can be processed by the list predicates `append/3`$_{[3.4]}$ and `length/2`$_{\text{ISO}}$.

C.6. Predicate `unify_with_occurs_check/2`

```
1  %% unify_with_occurs_check(?Term1, ?Term2)                    PROLOG
2  unify_with_occurs_check(X, Y) :-
3    X = Y,                                                       [3.2.1]
4    acyclic_term(X).                                             [ISO]
```

The predicate `unify_with_occurs_check/2`$_{\text{ISO}}$ performs the unification of two terms under consideration of possibly created cyclic variable bindings. In our implementation, the latter are tested with the help of the predicate `acyclic_term/1`$_{\text{ISO}}$ of the ISO Prolog standard. It checks that the term `X` does not contain cycles, i. e., the recursive processing terminates.

C.7. Predicate `otherwise/0`

```
1  otherwise. % simply always true                              PROLOG
```

In several code snippets we combine multiple of Prolog's *if-then-else* (with infix operators `->/2`$_{\text{ISO}}$ and `;/2`$_{\text{ISO}}$). The result is similar to *switch-case* statements in imperative programming languages. For better readability, we always use `otherwise/0` as the very last condition, following the notion of Haskell and the `default` clause used in most imperative programming languages.

C.8. Predicate `memberchk/2`

```
%% memberchk(?Elem, +List)                                    PROLOG
memberchk(X, [X|_]) :- !.                                     [3.2.2]
memberchk(X, [_|T]) :- memberchk(X, T).
```

In our implementations, we often use lists to represent named key-value associations (instead of using SWI-Prolog's dicts, cf. Section 3.3.5). `?- memberchk(Term, List)` unifies `Term` with the first matching element of the list `List`. Unlike the traditional alternative `member/2`$_{\text{SWI}}$, SWI-Prolog's built-in `memberchk/2` is semi-deterministic and does not backtrack after a successful unification for a found member.

C.9. Predicate `ord_intersection/3`

```
%% ord_intersection(+X, +Y, ?Z)                               PROLOG
ord_intersection([], _, []) :- !.                             [3.2.2]
ord_intersection(_, [], []) :- !.                             [3.2.2]
ord_intersection([X|Xs], [Y|Ys], [X|Zs]) :-
  X = Y,    % X and Y are the same                            [3.2.1]
  ord_intersection(Xs, Ys, Zs).
ord_intersection([X|Xs], [Y|Ys], Zs) :-
  lt(X, Y), % X is strictly smaller than Y
  ord_intersection(Xs, [Y|Ys], Zs).
ord_intersection([X|Xs], [Y|Ys], Zs) :-
  lt(Y, X), % given that X and Y are not equal
            %   and X is not strictly smaller than Y
            %   this could also be omitted
  ord_intersection([X|Xs], Ys, Zs).

% possible definition of lt/2
lt(X, Y) :-
  ( number(X) ->                                              [3.3]
```

```
19    X < Y          % for numbers                    [3.2.1]
20  ; X @< Y ).      % for atoms and strings          [5.1.4]
```

In Section 10.1.2, we use `ord_intersection(+X,+Y,?Z)` to calculate the ordered set representation `Z` of the intersection between the ordered sets `X` and `Y`. This predicate is autoloaded in SWI-Prolog as part of the built-in *library(ordsets)* [136, Sec. A.26].

Note that following its logical reading, `ord_intersection/3` could also be used with, e. g., `X` and `Z` given to generate possible solutions for `Y`. This would only be useful for a definition of `lt/2` that can also generate candidates.

C.10. Predicate `ord_memberchk/2`

```
1 %% ord_memberchk(+X, +List)                        PROLOG
2 ord_memberchk(X, [X|_]) :- !.                      [3.2.2]
3 ord_memberchk(X, [Y|T]) :-
4   lt(Y, X),                                        [C.9]
5   ord_memberchk(X, T).
```

The predicate `ord_memberchk/2` is a faster version of the classical $\texttt{member/2}_{\texttt{ISO}}$ or $\texttt{memberchk/2}_{\texttt{[C.8]}}$ for ordered sets. It fails as soon as a list element is reached that is higher than `X`. We use it in the definition of domains in Section 10.1.3. It is autoloaded in SWI-Prolog's *library(ordsets)* [136, Sec. A.26].

C.11. Predicates `maplist/`n

```
1 %% maplist(:, ?, ?, ?)                             PROLOG
2 :- meta_predicate maplist(3, ?, ?, ?).             [3.6]
3 maplist(_, [], [], []).
4 maplist(Goal, [X|Xs], [Y|Ys], [Z|Zs]) :-
5   call(Goal, X, Y, Z),                             [C.1]
6   maplist(Goal, Xs, Ys, Zs).
```

The family of meta-predicates `maplist/`n applies a goal, which is given as its first argument, on all tuples of elements of the lists given in the remaining arguments. They are autoloaded in SWI-Prolog's *library(apply)*. For the smallest value $n = 2$, `maplist/2` applies the goal on each list element. The listing shows

the implementation of `maplist/4`, which can be easily adapted for other arities by the addition or removal of the list arguments. For instance, the goal `?- maplist([1,B,3], [6,5,C], [A,7,7], plus)` applies the ternary predicate `plus/3` on each triple and computes the answer substitution `A = 7, B = 2, C = 4`.

C.12. Predicate `flatten/2`

```
%% flatten(+NestedList, -FlatList)
flatten([], []) :- !.
flatten([L|Ls], Flattened) :-
  !,
  flatten(L, LF),
  flatten(Ls, LsF),
  append(LF, LsF, Flattened).
flatten(L, [L]).
```

PROLOG

[3.2.2]

[3.4]

The predicate `flatten/2` converts a list of lists into its non-nested equivalent. It is autoloaded in SWI-Prolog's *library(lists)*, with similar definitions in all major Prolog systems. A naive approach to combine sequences into parse trees relies on this logically impure predicate, as discussed in Section 8.4.3. Though it results in a more complex transformation scheme, we therefore completely avoid the use of `flatten/2`, following the remarks of [136, Sec. A.21]: "Ending up needing `flatten/2` often indicates, like `append/3`[3.4] for appending two lists, a bad design."

C.13. Operator `./2`

```
:- op(100, yfx, .).
```

PROLOG

The infix operator `./2` is used since SWI-Prolog of version 7 to extract values and evaluate functions on dicts [129]. For a short introduction to dicts we refer to Section 3.3.5. Goals containing the `./2` operator are translated to calls of `./3`[C.14] by SWI-Prolog at compile-time using term expansion.

C.14. Predicate ./3

```
%% .(+Dict, +Function, -Result)                                    PROLOG
.(Dict, get(Key), Value) :-
  dict_pairs(Dict, _Tag, KV),                                      [SWI]
  memberchk(Key=Value, KV).                                        [C.10]
```

With the dot notation based on the infix operator ./2[C.13], it is possible to get the value of a key, but also to evaluate a function on a dict. Predefined functions are put/$\{1,2\}_{\text{SWI}}$ to add or replace new (list of) keys, and get/1_{SWI}, which is a silently failing alternative for value accessing via Dict.Key.

Because dicts are proprietary data structures in SWI-Prolog version 7 and higher, they cannot be implemented with only predicates of the ISO Prolog standard. In the code listing, we give an exemplary definition of ./3 which adds the SWI-Prolog's get/1 function for dicts. It makes use of memberchk/2[C.8] and dict_pairs/3_{SWI}, which maps a dict to an equivalent associative list.

C.15. Predicate and Operator :</2

```
%% +Select :< +From                                                PROLOG
:- op(700, xfx, :<).                                              [5.1]
Select :< From :-
  dict_pairs(Select, SelectP),                                     [SWI]
  dict_pairs(From, FromP),                                         [SWI]
  subset(SelectP, FromP).                                          [SWI]
```

Normal unification of two dicts with =/2_{ISO} fails if both dicts do not contain the same keys. With `Select :< From`, it can be tested if Select is a sub-dict of From, i. e., the unification is performed only for keys that are present in From. In our implementation, we use dict_pairs/2_{SWI} again to convert the dicts into their corresponding association lists, which are then compared using subset/2_{SWI}, which is part of SWI-Prolog's autoloaded *library(lists)*.

C.16. Operator >:</2

```
%% +Dict1 >:< +Dict2                                               PROLOG
:- op(700, xfx, >:<).                                             [5.1]
```

The operator `>:</2` defines a partial unification of two dicts, unifying only values of keys that are present in both. Values associated to keys that do not appear in the other dict are ignored. Unlike `:</2`[C.15], this predicate is commutative.

D

Index on Prolog Language Extensions

Tokenisation

Term Parsing

D.1. allow_unicode

```
:- op(600, xfx, ∈).
X ∈ Xs :- member(X, Xs).
```

Many Prolog systems, including SWI-Prolog, offer full Unicode support. However, following the ISO Prolog standard, Prolog's alphabet is traditionally composed only of the 7-bit US-ASCII character set, thus symbols like ∈ cannot be used anywhere in the Prolog source code – neither as part of a token, nor in source code annotations. The setting `allow_unicode(true)` enables the full Unicode character set in *library(plammar)*. Internally, we use SWI-Prolog's implementation of `char_type/2`$_{\text{SWI}}$ to get a character's type class. This allows, for instance, all answers for the goal `?- char_type(Char, prolog_symbol)` to denote an unquoted operator, so that ∈ could be used in an internal DSL. For external DSLs that rely on Unicode characters, this option is also required to process the contained data via, e. g., definite clause grammars.

D.2. allow_unicode_character_escape

```
unicode_num(\u2C6F, 0x2C6F).
```

Besides the literal appearance of Unicode characters, special characters can be represented by escape sequences beginning with `\u` (4-digit hexadecimal) and `\U` (8-digit hexadecimal). This notation follows the de-facto standard known from other programming languages, including JavaScript, and therefore simplifies their integration as an internal DSL.

D.3. allow_missing_closing_backslash_in_character_escape

```
equal(\xa, \xa\).
equal(\40, \40\).
```

According to ISO 6.4.2.1, the hexadecimal and octal character escape sequences start and end with a backslash symbol. This flag makes the trailing backslash optional, which follows the traditional definition the older Edinburgh standard and is supported by many modern Prolog systems as well. In addition, this aligns with the notation for hexadecimal and octal escape sequences from other languages, including C and JavaScript.

D.4. `allow_symbolic_*_char_*`

```
symbolic_chars('\c\e\s').
```

In ISO 6.4.2.1, some of the symbolic control characters (e. g., *carriage return* `\r`) are defined. This backwards compatible extension adds those missing from the 7-bit US-ASCII character set, as they could possibly part of strings and quoted atoms from external sources. In *library(plammar)*, the three additional symbolic control characters can be enabled separately by the following options:

- `allow_symbolic_no_output_char_c`
- `allow_symbolic_escape_char_e`
- `allow_symbolic_space_char_s`

D.5. `allow_shebang`

```
#!/usr/bin/env swipl
```

In many programming languages, it is possible to specify the program loader script path in the very first source code line. This character sequence is started by `#!` (called *shebang* or *hashbang*). The implementation of this language extension is presented in Section 9.3.2. Besides the actual usage for Prolog programs, the setting `allow_shebang(true)` simplifies the processing of an existing source code file from another programming language in case this language can be modelled solely as an internal DSL apart from the leading shebang.

This language extension is not backwards compatible, because `#!` can also be the legal start of a normal Prolog clause, e. g., after declaring `#/1` and `!/1` as prefix operators.[52] On the other hand, it has to be added at the tokenisation level, because the arbitrary character sequence following the shebang is not required to be a valid Prolog term.

D.6. `back_quoted_text`

```
javascript_template_string(`<h1>Hello, ${name}!</h1>`).
```

[52] Because of the special meaning of the cut token `!`, the character sequence `#!` does not constitute a valid Prolog operator name on its own.

Many programming languages suffer from the lack of a syntax for multi-line strings. If a string spans multiple lines, one often has to concatenate multiple strings. This has been resolved in some programming languages by introducing a new literal based on the back quote character `` ` ``, for instance in Java since version 13, JavaScript (cf. Section 4.3.3), and Go.

In Prolog, back quoted string literals have been defined and used from the very beginning. However, though the ISO Prolog standard defines their general format in its EBNF, originally they do not constitute a token – ISO 6.4.7 instead explicitly allows them as a valid language extension to denote a character string constant. This is enabled in *library(plammar)* by the `back_quoted_text` flag, which allows to integrate multi-line strings from other languages without further modifications.

As of today, back quoted string literals are supported by all major Prolog systems, though their interpretation differs. In SWI-Prolog, its term representation can be set via the `back_quotes` flag, which supports the same values as `double_quotes`, which we presented in detail in Section 3.3.4.

D.7. `var_prefix`

```
:- op(800, fx, SELECT).
```

As introduced in Section 3.1.1 and formally specified in the EBNF definition of the nonterminal *variable* in Figure 5.2, a Prolog variable starts either by an uppercase letter, or with the underscore as its prefix. With the setting `var_prefix(true)`, only the second form denotes valid variable names. Symbols beginning with an uppercase letter denote atoms. This setting simplifies the integration of case-sensitive external languages as internal Prolog DSLs. Obviously, it is not backwards compatible.

SWI-Prolog supports the module-wide flag `var_prefix` since version 7.3.27 [136, Sec. 2.12]. It is used in SWI-Prolog's RDF library [138], and for the integration of the statistical environment *R* [5].

D.8. `allow_digit_groups_with_space`

```
max_binary(0b1111 1111).
```

Sequences of integer tokens can never be part of a Prolog term, since numbers are not allowed to be declared as operators. Therefore, they can be defined in a backwards compatible manner as a new token that depicts a single integer written in digit

groups only for improved readability. Its implementation in the form of a finite-state machine is presented in Section 9.3.4. Digit groups are natively supported by SWI-Prolog [136, Sec. 2.16.1.5].

Note that digit groups with single spaces in-between are allowed only for integers of radix 10 or lower. For hexadecimal numbers, the term `0x0 f` is ambiguous, since `f` could also be declared as an operator, resulting in a valid Prolog term, too.

D.9. `allow_digit_groups_with_underscore`

```
max_hexadecimal(0xffff_ffff).
```

Besides using spaces, digit groups can be separated by single underscores between digits in a numerical literal. This also works with hexadecimal numbers, because the underscore would otherwise denote the anonymous variable token, which again cannot be defined as an operator, thus the character sequence `0x0_f` can be a valid Prolog term only using this flag. Separating large numeric literals into digit groups by underscores is supported by several other programming languages, including Java (since version 7) and JavaScript (proposal for inclusion in the ECMAScript standard). Therefore, this Prolog language extension allows their integration as internal DSLs.

D.10. `allow_integer_exponential_notation`

```
eps(1e-12).
```

Following ISO 6.4.5, a floating number is defined in the nonterminal `float_number_token//0`. It expects a fractional number including the decimal symbol `.`, optionally followed by the exponent. As introduced for *library(plammar)*'s finite-state machine in Section 9.3.4, this requirement is relaxed by some Prolog systems, which additionally allow the exponential notation immediately after an integer, avoiding the extra `.0` fractional part.

This shorter notation for exponential numbers is also supported by many other programming languages, including Java and JavaScript. Therefore, the language extension allows their integration as part of an internal DSL in Prolog. This modification of Prolog's syntax is not backwards compatible if we declare `e/2` as an infix operator, and similarly postfix operators like `e3`, `e4`, and so on. In this case, the character sequences `1e-12`, `1e3`, and `1e4` also natively constitute valid Prolog terms, but of different meanings.

D.11. `rational_syntax`

```
rational(1/3).
rational(1r3).
```

This new syntax for rational numbers as a primitive data type in Prolog has been added to *library(plammar)* mainly because of its common usage in modern Prolog programs. It is supported by ECLiPSe and SWI-Prolog from version 8.1.22 [136, Sec. 2.16.1.6]. Similar to `allow_integer_exponential_notation`[D.10], it is not backwards compatible for programs that expect the infix operators `//2`$_{\texttt{ISO}}$ and `r/2`, or the postfix operators of the form `r`n`/1` to work on integer arguments. On the other hand, it does not extend Prolog's expressiveness regarding DSLs, because structures of `//2` and `r/2` also natively constitute valid Prolog terms.

D.12. Additional Non-escaped Quote Characters

```
char_code(0'', 39).
char_code(0'<tab>, 9).
char_code(0'
, 10).
```

This extension provides a backwards compatible alternative notation for the ISO Prolog standard's character code constants `0'\'`, `0'\t`, and `0'\n`. It does not extends Prolog's expressiveness regarding other languages, but is frequently used in existing Prolog programs. In *library(plammar)*, the three additional non-quoted characters can be enabled separately by the following options:

- `allow_single_quote_char_in_character_code_constant`
- `allow_tab_as_quote_char`
- `allow_newline_as_quote_char`

D.13. `allow_infix_and_postfix_op`

```
:- op(500, fy, ++).
:- op(600, yf, ++).
:- op(600, xfy, ++).
f(++ x ++ x ++ x ++).
```

ISO 6.3.4.3 recommends to not declare the same name as an infix and postfix operator. This is justified by possible optimisations for the parser, which can immediately decide the specifier of an operator without too much look ahead. However, when reaching the clause's end, the term's structure is guaranteed to be unambiguous. For instance, the canonical term for our example is `++(++(x),++(x,++(x)))`.

D.14. `allow_compounds_with_zero_arguments`

```
run().
```

Since version 7, SWI-Prolog also supports compound terms of the form $a()$ that have no arguments [136, Sec. 5.3.2], but differ from the standalone atom a. The ISO Prolog standard instead requires compound terms to have at least one argument (ISO 6.3.3). The extension by SWI-Prolog originally aimed to allow functions on dicts (cf. Section 3.3.5). In addition, it results in a broader support for DSLs to be integrated internally. Most importantly, this allows to describe definitions and calls of functions without arguments.

D.15. `allow_empty_atom`

```
:- op(600, fx, function).
function ().
```

The ISO Prolog standard defines `{`, `}`, `[`, and `]` as dedicated tokens. Their pairs `{ }` and `[ ]` (with optional LTS in-between) constitute atoms (ISO 6.3.1.3). However, there is no equivalent for the empty pair of parentheses `( )`. It can be added to the ISO Prolog standard's EBNF by the following two grammar rules:

```
atom = open, close ;                                    EBNF DSL
atom = open_ct, close ;
```

The second case conflicts with the previously presented language extension `allow_compounds_with_zero_arguments`[D.14] for terms like `run()` (i. e., with empty LTS), if `run/1` is additionally also declared as a prefix operator. Both flags consume the `open_ct//0` token, but depict different parse trees:

```
?- prolog_parsetree(string("run()."), PT,                TOPLEVEL
   [ allow_compounds_with_zero_arguments(true),
     allow_empty_atom(true), operators([op(600,fx,run)])]).
PT = prolog([clause_term([term([atom(_), open_ct(_), close(_)])])]) ;
PT = prolog([clause_term([
```

```
term(fx, [op(atom(_)), term([open_ct(_), close(_)])])]) .
```

This unambiguity can be resolved by referring only to the nonterminal `open//0`, i. e., by using only the grammar rule of the aforementioned extension to the ISO Prolog standard's EBNF.

We successfully used this language extension for the modelling of XPath as an internal Prolog DSL, which we present in detail in Section 10.4. The need for such a language feature was also noted before in [130].

D.16. allow_curly_block_op

```
:- op(100, xf, {}).
:- op(600, xfx, while).
do { writeln(X) } while (member(X, [1,2,3])).
```

Without further modifications, both `{ }` and `{ t }` are valid Prolog terms according to the ISO Prolog standard. The empty curly brackets constitute a single atom (ISO 6.3.1.3) with no special meaning. The character sequence `{ t }` on the other hand denotes the *curly bracketed term* `{}(t)`, i. e., a compound term of functor `{}/1` (ISO 6.3.6).

In most imperative programming languages, curly brackets are used to enclose code blocks, e. g., in the definition of classes, functions, or loop bodies. To support the internal integration of these DSLs, we additionally allow to use the atom `{}` as a *block operator*. It is typically declared as a postfix operator with type `xf` and low precedence, but can be equally defined as any other type. In the canonical representation of the compound term, the block's inner content is added as its first argument. This is why it is of functor $\texttt{\{\}}/\{2,3\}$, i. e., with an arity of 2 in case of a postfix or prefix operator, and an arity of 3 when defined as an infix operator, unlike the traditional arities known from operators. For instance, given that `{}` is defined as a postfix or prefix operator, the terms $\texttt{s}\cdot\texttt{\{ t \}}$ and `{ t } s` are valid and read as `{}({t},s)`; for an infix operator, $\texttt{s}_l\cdot\texttt{\{ t \}}\ \texttt{s}_r$ results in the compound term $\texttt{\{\}(\{t\},s}_l\texttt{,s}_r\texttt{)}$. In this canonical representation, the block operator's inner term `t` is additionally stated as the curly bracketed term `{}(t)`.

Note that the curly block operator notation is actually not an extension but a deviation of the ISO Prolog standard, which in Technical Corrigendum 2 disallows to declare the curly bracket pair as an operator (ISO 6.3.4.3). In addition, the notation of block operators conflicts with the proposed language extension for dicts (cf. Appendix D.18). If `{}/1` is defined as a postfix or infix operator, the term `a{ b: c }`

is ambiguous. We therefore propose to always use the LTS in front of the opening curly bracket. With the flag `dicts`[D.18] disabled, the term is unambiguously parsed as `{}({:(b,c)},a)`, even without a preceding LTS.

Introducing block operators was proposed by José F. Morales. It was discussed for inclusion in the ISO Prolog standard, but because of too many conflicts with existing extensions no agreement was reached [136, Sec. 5.3.3]. In our *library(plammar)*, this language extension can be activated by the flag `allow_curly_block_op`. Block operators are also supported by SWI-Prolog and YAP. In SWI-Prolog, it is always enabled, because it takes effect only after the definition of the block operator by the user.

D.17. `allow_square_block_op`

```
:- op(100, xf, []).
fib[N-2] + fib[N-1] >> fib[N].
```

Besides postfix curly bracketed terms to enclose code blocks, imperative programming languages often use postfix terms in square brackets to denote array subscripting. Similar to the previously presented option, the flag `allow_square_block_op` enables declarations of the atom `[]` as an operator in *library(plammar)* and SWI-Prolog. It is typically declared as a postfix operator with low precedence. In contrast to the curly block operator's `xf`-type, array subscription is typically used as an `yf`-operator to allow nested array access. Given this postfix operator declaration, `t[`s_1`][`s_2`]` produces the compound term `[]([`s_2`],[]([`s_1`],t))` in canonical notation. Similar to the curly block operator, the block's inner terms s_1 and s_2 are additionally each stated in a classical list. By disallowing the definition of `[]` as an operator (ISO 6.3.4.3), `t[`s_1`][`s_2`]` cannot be parsed as a valid Prolog term.

D.18. `dicts`

```
data(alice, _{ birth: 1986 }).
```

In Section 3.3.5, we introduced *dicts* as SWI-Prolog's primary data type for named key-value associations. Their syntax resembles JSON and the notation for associative arrays known from other programming languages. However, these objects can often be recreated in an internal Prolog DSL by only using Prolog's built-in support for curly bracketed terms (ISO 6.3.6). To additional allow the specification of a leading tag, the flag `allow_curly_block_op`[D.16] could be used equally.

The flag `dicts` therefore mainly intends to create parse trees in *library(plammar)* that reflect the structure of named key-value associations. Similar to the ISO Prolog standard's definition of lists with the nonterminals `arg_list//0` and `arg//0`, we defined the nonterminals `key_value_list//0` and `key_value//0` to process a dict's content.

D.19. `allow_operator_as_operand`

```
sign(- * -, +).
```

Following ISO 6.3.1.3, any declared operator constitutes a term with a priority of 1201. This ensures that if an operator is used as an operand, it has to be put in parentheses. By relaxing this constraint and treating operators as terms with a priority of 0 as well, character sequences like `X = -` become valid again. This is the default behaviour of SWI-Prolog, but can be disabled by setting the program flag `iso(true)`.

D.20. `allow_arg_precedence_geq_1000`

```
bitwise_or(A | B, Ored) :- Ored is A \/ B.
test(car->colour, red).
```

As introduced in Section 10.1, arguments of compound terms as well as list elements are expected by ISO 6.3.3 to be of a priority less than 1000. This ISO Prolog standard compliant behaviour requires additional parentheses around terms of the built-in operators `->`$/2_{\texttt{ISO}}$, `;`$/2_{\texttt{ISO}}$, `|`$/2_{\texttt{SWI}}$, `-->`$/2_{\texttt{ISO}}$, and `:-`$/\{1,2\}_{\texttt{ISO}}$. By lowering this restriction, internal Prolog DSLs are allowed to omit these.

SWI-Prolog by default allows arguments of any priority. It can be disabled by setting the program flag `iso(true)`.

D.21. `allow_variable_name_as_functor`

```
COUNT(*).
```

Every compound term's functor name is required to be an atom by the ISO Prolog standard. This requirement can be relaxed to include variable names to be compatible with more DSLs. Its interpretation is left to the Prolog system. SWI-Prolog automatically converts the functor name to the corresponding atom when reading in compound Prolog terms of this form, so our example is identical to `'COUNT'(*)`.

Note that the given example is also valid Prolog by using the tokenisation setting `var_prefix`[D.7] instead, which similarly handles `COUNT` as an atom. While `allow_variable_name_as_functor` preserves the general notation of variables, it is not as flexible as `var_prefix`[D.7], since it affects only the notation of compound terms in functional notation. It is still not allowed to define `COUNT` as an operator without enclosing it in single quotation marks.

D.22. `allow_unquoted_comma`

```
two_commas([(,), (,)]).
two_commas([,,,]). % with allow_arg_precedence_geq_1000[D.20]
```

The comma in Prolog has a special meaning and is defined by the ISO Prolog standard as a separate token. It therefore does not constitute neither a name nor a term. Thus, `?- X = (,)` fails – even though the additional enclosing in parentheses avoid possible problems with respect to the term's priority. Following the ISO Prolog standard, the comma can be used only as the built-in infix operator, or as the atom `','` in single quotation marks. This flag enables its usage as term, so it can be literally used as argument in compound terms and lists. Together with `allow_arg_precedence_geq_1000`[D.20], this allows for terms like `[, , ,]`, though its practical benefits with respect to the integration of DSLs remain limited. In SWI-Prolog, this behaviour can be disabled by setting the program flag `iso(true)`.

D.23. `allow_dot_in_atom`

```
:- op(600, xfx, is.a).
```

The full stop `.` constitutes a separate token in Prolog, which is usually used to mark the end of a clause. It can additionally be defined as an operator. This allows cascading function notations as known from other programming languages. With `./2` as an infix operator of type `yfx`, `car.colour.rgb` constitutes the canonical compound term `.(.(car, colour), rgb)`.

SWI-Prolog suggested to alternatively change the grammar for the nonterminal `name_token//0`. The transition rules for the state `seq_alphanumeric_char` in our lexer (cf. Table 9.2) therefore also accept dots embedded into atoms. The dot must be followed by an identifier continuation character, i. e., a letter, digit or underscore. The dot is allowed in identifiers in many languages, which makes `allow_dot_in_atom` a useful flag when integrating DSLs, in particular as it paves

the way for operators with embedded dots. This language extension is only backwards compatible for programs that neither use `./2` as an infix operator, or rely on atoms with embedded dots as operators.

D.24. `allow_implicit_end`

```
fact
```

Many languages can be defined as internal Prolog DSLs. But although they are of manifold shape due to their declaration of different operators, all these internal DSLs have one thing in common: they are required to end with `.` as the (possibly single) clause's `end//0` token. In our definition of EBNF as an internal Prolog DSL in Section 5.6, we used the classical symbol `;` to delimit individual grammar rules but the very last, which instead ends with the end token.

According to Iso 6.2, a Prolog program consists of *Prolog text*, which is a sequence of the nonterminal `p_text//0`. Listing 9.6 shows its original grammar from the ISO Prolog standard. We propose a second, alternative grammar rule for `prolog_text//0`:

```
prolog_text = term ;                                    Ebnf Dsl
```

This way, the overall source code program only needs to be a valid Prolog term. On the other hand, the addition for the nonterminal `prolog_text//0` instead of the self-referential `p_text//0` ensures that we do not mix classical Prolog clauses with those without the trailing end token. Which alternative has to be applied can be determined in constant time and requires no backtracking. If the lexer returns `.` as its very last token, the classical definition is used, and otherwise our alternative proposal.

List of Figures

List of Tables

List of Listings