Prolog Coding Guidelines: Status and Tool Support

Falco Nogatz
University of Würzburg, Germany
falco.nogatz@uni-wuerzburg.de

Philipp Körner
University of Düsseldorf, Germany
p.koerner@hhu.de

Sebastian Krings
Niederrhein University of Applied Sciences, Germany
sebastian.krings@hs-niederrhein.de

The importance of coding guidelines is generally accepted throughout developers of every programming language. Naturally, Prolog makes no exception. However, establishing coding guidelines is fraught with obstacles: Finding common ground on kind and selection of rules is matter of debate; once found, adhering to or enforcing rules is complicated as well, not least because of Prolog's flexible syntax without keywords. In this paper, we evaluate the status of coding guidelines in the Prolog community and discuss to what extent they can be automatically verified. We implemented a linter for Prolog and applied it to several packages to get a hold on the current state of the community.

1 Introduction

The importance of coding guidelines is generally accepted throughout developers of every programming language. Naturally, Prolog makes no exception. Hence, different attempts at establishing coding guidelines for Prolog have been made. The most recent work towards unified coding guidelines for Prolog has been made by Covington et al. (2012) as well as McLaughlin (1990). Further comments and additions to McLaughlin's style guide have been posed by Kaplan (1991), who focused on the presentation and implementability of the given rules.

However, establishing coding guidelines is fraught with obstacles: Finding common ground on kind and selection of rules is matter of debate; once found, adhering to or enforcing rules is complicated. Furthermore, not all rules deemed sensible can be automatically verified.

Covington et al. suggested the usage of automated tools to improve readability [1, Section 2.17]. While the authors suggested numerous tools able to reformat source code, we want to go move beyond reformatting and automatically verify a greater extent of the coding guidelines suggested. In order to provide such a tool, we evaluate the status of coding guidelines in the Prolog community, discuss to what extent they can be automatically verified and suggest possible ways to move forward. We try to assert the impact of existing guidelines empirically and enquire if there is consensus on how to format Prolog code.

Following, in Section 2, we discuss how well the rules introduced by Covington et al. can be checked automatically. In particular, we define a subset we deem suitable for automatic verification and discuss why others cannot be checked by a software tool. For the ones where automation is feasible, we implemented a style checker and linter which we introduce in Section 3. The linter is applied to a selection of publicly available Prolog files from multiple sources. In Section 4, we present the current state of coding guidelines and to what extent the community adheres to them. Finally, in Sections 5 and 6 we discuss related and future work and conclude.

Submitted to:
ICLP 2019

Table 1: Implementation Status of COV Rules

Category	Implementable	Not-Implementable	Implemented
Layout	2.1 – 2.9, 2.14 – 2.17	2.10 – 2.13	2.1 – 2.7
Naming conventions	3.1, 3.4, 3.10, 3.12	3.2, 3.3, 3.5 – 3.9, 3.11, 3.13, 3.14	3.1, 3.4
Documentation	4.1	4.2 – 4.5	-
Language idioms	5.3	5.1, 5.2, 5.4 – 5.13	-
Development, debugging, and testing	6.5, 6.7	6.1 – 6.4, 6.8 – 6.18	-

2 Applicability of Automatic Checking to Coding Guidelines

The coding guidelines given by Covington et al. (2012) and McLaughlin (1990) involve different aspects of a Prolog program, ranging from formatting instructions to naming of variables and predicates as well as advice on implementation aspects such as the usage of cuts. In the following, we discuss some rules stated in [1]. For the sake of simplicity, we refer to them as COV, with COV 2.1 representing the coding guideline of Section 2.1. Table 1 lists which rules can be automated and to which extend they are already implemented in our linter. To give an impression of why rules can or cannot be automated, we discuss some of them below.

Certain rules can easily be implemented, in particularly those involving source code layout. COV mentions indentation with spaces instead of tabs, consistent indentation, limits regarding the length of lines or the number of lines in a predicate as well as the use of spaces and newlines. In addition, COV advises a decision on consistent ways to write if-then-else and disjunction. However, even certain layout rules cannot easily be verified automatically since they rely on personal opinion or subjective assessment, for instance:

- Use layout to make comments more readable (COV 2.10).
- Avoid comments to the right of a line of code, unless they are inseparable from the lines on which they appear (COV 2.11).

We argue that both rules are unsuitable for automatic linting. Readability is a highly personal evaluation, depending on programmer's taste and not evaluable without understanding a comment's content. While language processing techniques might be able to capture parts of intention and mood using, for instance sentiment analysis [12], we see no appropriate technique for incorporation in a linting tool.

Similar problems occur when trying to verify naming conventions (COV 3). While we can of course check the consistent use of underscore-style or camel case as well as the consistent naming of identifiers, we again face rules which are not easily implementable:

- Rules regarding the pronunciation of identifiers (COV 3.2, 3.3).
- Rules taking into account the semantics of a predicate, such as assigning certain names to predicates representing relations vs. those to be understood procedurally (COV 3.6, 3.7),
- Add the types on which a predicate operates to the predicate name (COV 3.13).

While again language processing techniques might be able to solve the first problem, capturing the intended semantics of a predicate is complicated at best. Since Prolog is an untyped language, the types on which a predicate operates cannot be computed in general. Ciao Prolog includes an assertion language that allows annotating Prolog programs with modes, types and further information that could be used to

improve coverability of rules [18, 17]. In other cases, an external library such as *plspec* [9] for specifying types or other constraints on variables could be integrated into a style checker.

Rules regarding availability of documentation (COV 4) can only be verified with respect to the existence of documentation and comments. However, enforcing the existence of comments might just drive programmers to add empty or meaningless comments.

The guidelines in COV 5 discuss language idioms and highlight the importance of certain ways to write predicates, going into detail on techniques such as tail recursion. While these usually involve common patterns that can be checked syntactically, deciding on when to use them remains impossible for a style checker. This holds true for all rules regarding language idioms:

- Deciding whether a predicate is steadfast is impossible if we aim for absolutely correct classification. However, a common pattern that often collides with steadfastness is the unification of a parameter after a cut. In that case, it is impossible to backtrack into another rule that might be more appropriate for the given parameter.
- Without further annotations regarding type or mode of inputs, it is impossible to decide whether input arguments have been placed before output arguments. Again, we refer to the different articles regarding type systems and mode checkers for Prolog [15, 6, 9].
- Deciding whether a cut is red or green is impossible in general. Hence, a linter cannot use different rules for red and green cuts as suggested by COV 5.4.
- COV 5.7 and 5.8 suggest avoiding `append/3` as much as possible for performance reasons. However, we do not think its sensible to mark its usage as stylistic errors in general. The same holds true for `asserta/1, assertz/1` and `retract/1` as COV 5.10 discusses.
- Of course a linter can mark occurrences of body recursion and suggest to use tail recursion (COV 5.9). At the same time, we cannot estimate whether a reimplementation will be worthwhile.
- Operating in badges (COV 5.11) and using built-in sorting algorithms rather than custom algorithms (COV 5.12) both describe situations not easily detectable by our analysis, e.g., our linter would have to figure out if a certain predicate indeed implements quicksort.
- Augmenting Prolog with a run-time type and mode checking system as suggested in COV 5.13 is a point we absolutely share. However, this again should not be part of a linter. Instead, we refer to the type systems suggested [20] .

The guidelines in COV 6 focus on the general process of software development and testing and are thus less appropriate for source code analysis. Two of them however can easily be checked automatically: avoiding constructs that are almost always wrong (COV 6.5) and detecting and avoiding magic constants (COV 6.7).

To summarize, the current selection of coding guidelines available for Prolog can only partially be verified and enforced automatically. In particular, rules involving sense and meaning of comments or pronunciation cannot efficiently be verified. To enforce the automatically verifiable rules, we implemented a linting tool which we will discuss in the following section.

3 Static Source Code Analysis for Prolog

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, e.g., because

misspelled variable names are never used, and highlight means for optimizations, e.g., to eliminate loop invariants. In addition, static source code analysis is used to enforce the adherence to coding standards.

In contrast to imperative programming languages with a fixed set of keywords, Prolog's syntax is very flexible. With self-defined operators, terms can be written without parentheses. For instance, the term *"a b"* in functional notation represents either the compound *"a(b)"* or *"b(a)"*, depending on whether `a/1` is a prefix operator or `b/1` is a postfix operator. For code analysis this distinction is required, because some coding guidelines (e.g., COV 3.5 and 3.6) target predicates but not arguments. Since the used operators could also be defined in imported modules, static analysis of Prolog programs often requires information collected at runtime. Some Prolog systems, most notably Ciao Prolog, use refined module systems to allow for more thorough static analysis [3].

For Prolog systems in general, this information is available on compilation. Therefore, a feasible approach to implement a linter for Prolog might be to use its built-in term expansion mechanism. In the following, we discuss this technique as well as the traditional way to implement a linter using a parser.

Term Expansion. Term expansion is a mechanism to rewrite Prolog code at compilation, similar to macros in other programming languages. Although not part of the ISO Prolog standard, it is supported by most Prolog systems. When loading code into Prolog, its compiler calls `expand_term/2` on each term read from the input. The asserted term can be modified by defining the predicate `term_expansion(+Term1,-Term2)` taking the original `Term1` and replaces it by `Term2`. Both take account of operator definitions, i.e. functional notation is handled by the Prolog compiler.

Without modification of the term, `term_expansion/2` provides a simple way to access all clauses on compilation. Hence, it can be used to check adherence to the coding guidelines of COV 3 and 5, related to naming conventions and language idioms. Layout (COV 2) cannot be checked this way, because only the Prolog terms are given to `term_expansion/2`; all styling information is omitted, since it is not needed for compilation.[1] Source code annotations cannot be accessed in the term expansion. Although SWI-Prolog allows to define the hook `prolog:comment_hook/3` [27, Sec. B.8] to get access to all comments, the tests regarding source code annotations (COV 4) would be split across this hook and the term expansions.

Another disadvantage of linting using term expansion is that `term_expansion/2` is stateless. Several coding conventions require knowledge about other clauses, which one would normally pass as an additional argument. Instead, persistent information would have to be handled using dynamic predicates.

With term expansion, parsing is left to the compiler of the used Prolog system.[2] This way, the linting rules can be defined by just processing Prolog terms, without having to implement a complete Prolog parser.

Parser. Even though execution is not needed, term expansion can only be used to analyze code in valid Prolog syntax with respect to the used Prolog system. Many (all?) Prolog systems diverge from the

[1] In SWI-Prolog and Quintus, the option `subterm_positions` of `read_term/2` provides only character positions which do not allow to correctly identify newlines. On the other hand, SICStus' `layout` option provides only the line numbers of the subterms but not their actual indentations. The original variable names can be accessed in all implementations, because the needed option `variable_names` is part of the ISO Prolog standard.

[2] For automatic source code analysis – for example in our empirical study in Section 4 –, often source code from unknown sources is tested. To prevent security issues when loading potential malicious code, we strongly recommend using the empty list `[]` for `Term2` in `term_expansion/2`. As a result, external code is neither asserted nor executed.

$$
\begin{aligned}
\textit{variable token} &= \textit{anonymous variable} \mid \textit{named variable} \ ; \\
\textit{anonymous variable} &= \textit{variable indicator char} \ ; \\
\textit{named variable} &= \textit{variable indicator char}, \ \textit{alphanumeric char}, \ \{ \ \textit{alphanumeric char} \ \} \\
&\mid \ \textit{capital letter char}, \{ \ \textit{alphanumeric char} \ \} \ ; \\
\textit{variable indicator char} &= \textit{underscore char} \ ; \\
\textit{underscore char} &= \text{"\_"} \ ;
\end{aligned}
$$

Figure 1: Grammar rules for a variable token in Prolog as defined in the ISO Prolog standard.

ISO Prolog standard.[3] In consequence, support for multiple Prolog dialects and different Prolog systems cannot be achieved relying on term expansion, as it is restricted to the language features provided by the used system. The same holds true for reusing existing parsers of different Prolog interpreters. In contrast, with a dedicated Prolog parser, even source code designed for a different target system can be analyzed.

Although the parser loses runtime information available in the term expansion approach – most importantly operator definitions – , we opted for this choice. In the following, we present the linter's architecture, including a feature-rich Prolog parser.

3.1 *library(plammar)* – A Prolog Parser and Serializer

Static analysis of source code is typically split into two phases: (i) the lexical analysis, that converts a sequence of characters into a list of tokens, and (ii) the combination of tokens in order to generate a structural representation, often in form of a syntax tree. In a third step, the concrete syntax tree (CST) is converted into an abstract syntax tree (AST), while removing and analyzing all layout information.

Lexical Analysis. The syntax of Prolog is specified in the ISO Prolog standard [5] using grammar rules in extended Backus–Naur form (EBNF). Nogatz et al. defined EBNF as an internal domain-specific language in Prolog [16]. Similar to DCGs, the grammar rules are translated into Prolog predicates with two additional arguments to hold lists of consumed resp. remaining symbols. In contrast to the traditional term expansion scheme of DCGs, the resulting Prolog predicates also contain an additional argument to automatically store the corresponding parse tree, based on the rule's nonterminal.

As an example, the EBNF grammar rules for a variable token as defined in [5, Sec. 6.4.3] are given in Figure 1. It is translated into the predicate `variable_token/3`, that creates a parse tree for a given input string and vice-versa. In Figure 2, we present an example query for the input string "`_a`".[4] The syntax of other Prolog tokens is defined similarly. As a result, our tool *library(plammar)* provides the predicate `prolog_tokens(?Source,?Tokens)` that takes Prolog source code and generates the list of tokens, and the other way around.

[3] *Conformity Testing I: Syntax*, list of ISO compliance for popular Prolog systems, collected by Ulrich Neumerkel: `http://www.complang.tuwien.ac.at/ulrich/iso-prolog/conformity_testing`.

[4] The ISO Prolog standard requires that a token shall not be followed by characters such that concatenating the characters of the token with these characters forms a valid token, i.e. only the second solution of Figure 2 is valid. This requirement is realized in other parts of our parser.

```
?- variable_token(Parse_Tree, "_a", R).
% first solution, consuming only "_":
R = "a", Parse_Tree = variable_token(anonymous_variable(
                        variable_indicator_char(underscore_char('_'))));
% second solution, consuming the whole string "_a":
R = "",  Parse_Tree = variable_token(named_variable([
                        variable_indicator_char(underscore_char('_')),
                        alphanumeric_char(alpha_char(letter_char(
                          small_letter_char(a) ))) ])) .
```

Figure 2: Usage example for the generated `variable_token/3` with the input string `"_a"`.

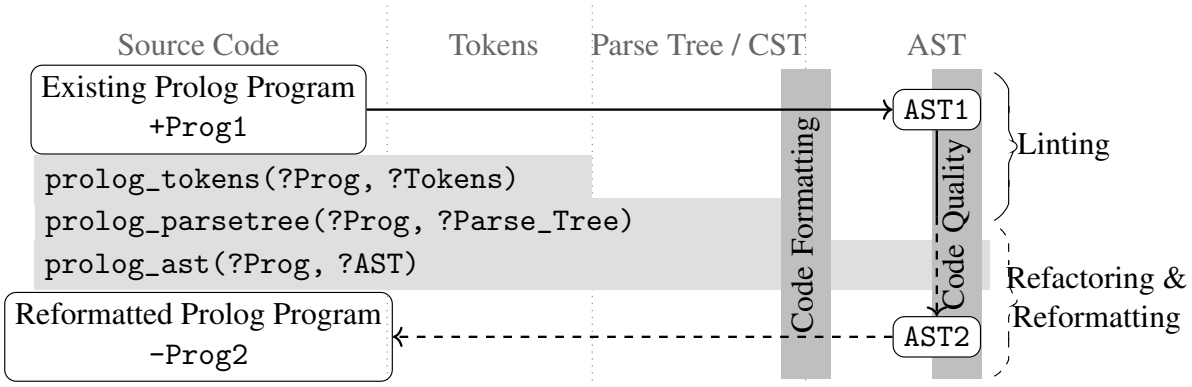


Figure 3: Architecture Overview for Linting with *library(plammar)*.

Parsing. The ISO Prolog standard specifies how to combine tokens into valid terms with EBNF again. Therefore, the list of tokens generated by `prolog_tokens/3` can be processed as before, consuming tokens rather than characters. The generated parse tree strictly follows the structure of the applied grammar rules, with its nonterminals as subtrees. [16] provide a more detailed discussion of aspects when parsing Prolog code this way, e.g., to correctly handle operator precedences.

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. On the other hand, this concrete syntax tree (CST) is very verbose. The contained layout information are only required to test the adherence to Cov 2; all other coding guidelines are applicable to an abstract syntax tree (AST). We therefore define a Prolog predicate that transforms a parse tree into its corresponding AST, and the other way around. For instance, a program with just a single rule returns the following AST:

```
?- prolog_ast(string("positive(X) :- X > 0."), AST).
AST = prolog([ rule(                                          % single rule
                compound(atom(positive), [variable('X')]),          % head
                [infix(>, xfx, variable('X'), integer(0))] )]).     % body
```

3.2 Rules for Code Formatting and Code Quality

Figure 3 summarizes the parser's overall architecture. For each step, *library(plammar)* provides a predicate: (i) `prolog_tokens/2` holds the tokens for the given Prolog source code;

(ii) `prolog_parsetree/2` generates the corresponding parse tree; and (iii) `prolog_ast/2` additionally transforms the CST into AST. Every predicate can take an additional argument to pass options, e.g., to specify the targeted Prolog system or dialect.

In general, we distinguish between *code formatting rules* and *code quality rules*. The first are related only to the CST; changes are not represented in the program's AST, code formatting rules are checked on-the-fly in the tree transformation step of `prolog_ast/2`.

Most Prolog tokens are allowed to be preceded by a *layout text sequence* [5, Sec. 6.4.1], i.e., a sequence of whitespaces, newlines, or comments. In the transformation step from CST to AST, this layout information is removed but tested to its adherence to the given guidelines. The preferred formatting can be specified as a list of options, also referred to by the coding guidelines by Covington et al.:

- COV 2.1 and COV 2.2: *indent(Integer)*,
- COV 2.3: *max_line_length(Integer)*,
- COV 2.4: *max_subgoals(Integer)* and *max_rule_lines(Integer)*,
- COV 2.5: *space_after_arglist_comma(yes/no)*,
- COV 2.6: *newline_after_clause(yes/no)*,
- COV 2.7: *newline_after_rule_op(yes/no)* and *newline_after_subgoal(yes/no)*,
- COV 2.14: *indent_between_repeat_cut(yes/no)*

Additionally, we provide options for common best-practices, e.g., to prevent trailing whitespaces.

In contrast to formatting rules, code quality rules consider only the AST. This distinction is also reflected in the implementation: code quality rules are defined separately as a tree traversal for the AST. Currently, they are simply checks, without further transformations. We support options to handle COV 3.1, COV 3.4 and COV 3.12.

COV 4 suggest style guidelines for source code comments, COV 5 and COV 6 recommend the usage and avoidance of several language idioms. These are not yet common in the Prolog community, though we want to add checks for them in the future.

3.3 Benefits from Prolog Features

Our implementation shows that Prolog is particularly suited for implementing linters and software analysis tools in general. In particular, the following features came in handy for us:

- *Definite clause grammars.* The definition of Prolog's syntax with EBNF rules resembles DCGs. As a result, the grammars for tokens and terms can be directly adopted from the ISO Prolog standard, resulting in an executable Prolog program after minor adjustments.
- *Backtracking.* For performance reasons, the parser avoids backtracking where possible; all grammar rules require at most a small look-ahead. Only in cases where the program's environment is underdetermined, e.g., if the targeted Prolog dialect is unknown, or operator definitions are missing, backtracking can be used. This way, the program "*a b.*" can be parsed, deducing that `a/1` has to be defined as prefix operator, or `b/1` as postfix operator.[5]
- *Logic Variables.* The settings we presented in Section 3.2 also support logic variables as arguments. As a result, it is bound to the correct value where possible. For instance, a variable in *max_line_length* is bound to the maximum value of all observed line lengths.

[5] In our empirical research of Section 4, we deactivated this option to deduce missing operator definitions, as it slows down the parsing process because of the bigger search space.

– *Predicates as relations.* Each predicate of *library(plammar)* expresses a relation between Prolog source code and its corresponding list of tokens, parse tree, or AST. For this purpose, the term expansion creating Prolog predicates based on EBNF was designed to use only pure predicates. *library(plammar)* can therefore be used in both directions without modification.

Currently, violations of coding guidelines are returned as a list of warnings or just printed when calling `prolog_ast/3` (code formatting rules) and `check_ast/2` (code quality rules):

```
?- prolog_ast(file('/tmp/in.pl'), AST, Options), check_ast(AST, Options).
```

The fact that *library(plammar)* can be used in reverse simplifies the extension to repair of rule violations. In this case, one just has to provide `Parse_Options` with non-strict settings for parsing, and `Serialize_Options` with the preferred style settings for serializing. Instead of `check_ast/2`, a predicate `transform_ast/3` has to be used that transforms the program's AST according to the coding guidelines:

```
?- prolog_ast(file('/tmp/in.pl'), AST_In, Parse_Options),
   transform_ast(AST_In, AST_Out, Serialize_Options),
   prolog_ast(file('/tmp/out.pl'), AST_Out, Serialize_Options).
```

3.4 Support for SWI-Prolog Extensions

We opted for a dedicated Prolog parser to support different Prolog systems and dialects. While we wanted to support as many Prolog systems as possible, we were particularly interested in SWI-Prolog. With the release of version 7, SWI-Prolog further diverged from the ISO Prolog standard [26]. We implemented 20 options to enable or disable non-ISO-conform language features supported by recent versions of SWI-Prolog. Among others, *library(plammar)* considers the following settings, each as *yes/no*:

– *dicts*: This adds a new token for structures with named arguments, e.g., "`point{ a: 1 }`". In addition, the operator "`.`" is added for field extraction.

– *allow_compounds_with_zero_arguments*: Allow compounds without arguments, e.g., "`a()`".

– *allow_arg_precedence_geq_1000* and *allow_operator_as_operand*: SWI-Prolog allows for terms that would require parenthesis according to the ISO Prolog standard, e.g., "`X = -`", and "`[a :- b]`" instead of "`[(a :- b)]`".

– *allow_integer_exponential_notation*: The ISO Prolog standard requires one to write "`1.0e3`" instead of "`1e3`".

The same approach could be followed to adapt to further Prolog systems and dialects. In our static code analysis, we only consider the program's syntax. Although possible, we do not yet check for system-specific built-in predicates.

4 Status Report

Even though there were several pleas for the consistent usage of coding guidelines in the Prolog community, it remains unclear, whether there is consensus on how to handle formatting and specific coding constructs. Several possible scenarios come to mind:

1. Most of the community adheres to one batch of guidelines, e.g., to Covington et al. (2012).

2. The community is fractured, adhering to different guidelines. One partitioning might be, that groups using SWI-Prolog write code differently than groups using SICStus. Another possibility might be that certain projects, e.g., packages shipped with SWI-Prolog, enforce a set of rules concerning the code.
3. Programmers have an individual, consistent coding style that they stick to.
4. Individual programmers do not care and mix different coding styles in their own projects.

We suspect that contributors working on larger, serious projects mostly adhere to a project-wide style, though there are some minor deviations to be expected. In contrast, individuals managing a small code basis, e.g., homework repositories, usually are not concerned with code style, mostly because they have not considered any guidelines or the project is not serious enough.

In the following, we use our tool to determine whether the Prolog community adheres to any guidelines and which of our hypotheses will be falsified when checked against reality.

4.1 Empirical Research

For our analysis, we consider the SWI-Prolog "batteries-included" packages[6] as well as the list of known SWI-Prolog packages[7]. A similar analysis for other dialects such as SICStus or Ciao is possible using the same grammar and linter as discussed above and will part of our future research. We discard unavailable packages and files only consisting of facts.

For all files, we set a 10-second timeout. We excluded source files larger than 1 MB or longer than 20.000 lines of code, as they are usually generated by some tool. Additionally, we excluded some packages due to inconsistent operator definitions within the package, invalid Prolog code, and operator definitions that are not supported any longer by SWI-Prolog.[8]

Community Packages. The considered code base consists of 3815 Prolog files from 251 packages, of which 2775 contained rules. More than 90 % of the files could be parsed by our tool. We elaborate on why others could not Section 4.3.

SWI Packages. Packages shipped with SWI-Prolog consist of 688 Prolog files, 609 of which could be processed and 530 containing at least one rule. They stem from 34 packages and consist of 137051 lines of code.

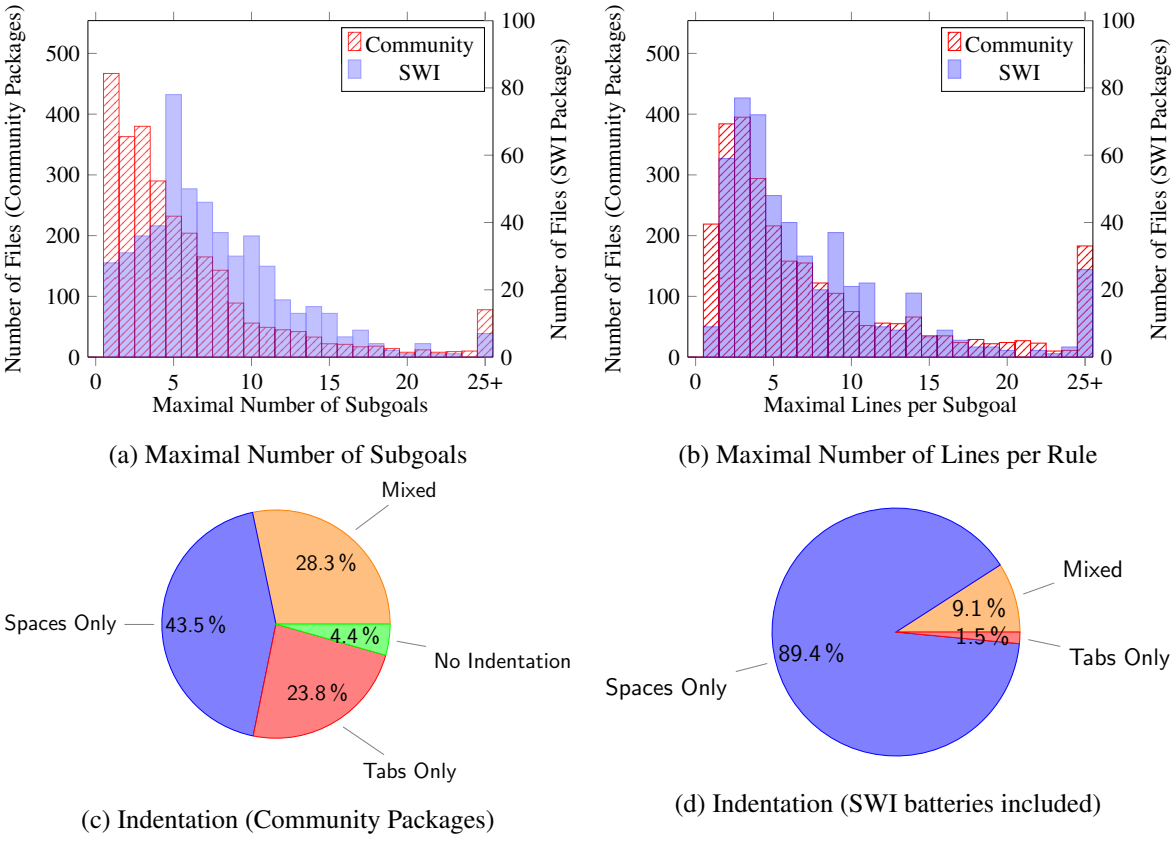
Following, we present the results for some coding guidelines (Cov 2.1, 2.2) in detail. Among the most discussed is how to properly indent: Figs. 4c and 4d show the share of different styles, i.e., no indentation vs. indentation with spaces or tabs that is consistent in the sense that spaces and tabs are not mixed, but not that the indentation size is the same. Lastly, both tabs and spaces might be used for indentation. As can be seen, there is no clear trend.

Limiting the horizontal length of source code lines is considered a best practice in Cov 2.3. From 361782 source code lines overall (including blank lines and comments), only 12431 lines (3.4 %) are longer than 80 characters. However, violations of this rule can be found in 879 files (31.6 %). In the "batteries included" packages, this figure is slightly smaller, with 756 out of 137051 lines (0.6 %) in 92 of the 530 (14.0 %) analyzed files exceeding 80 characters.

[6] `https://github.com/SWI-Prolog/swipl-devel/tree/9afb9384bb/packages`

[7] `http://www.swi-prolog.org/pack/list`

[8] A list of tested packages is available at `https://github.com/fnogatz/plammar-community-evaluation`.



(a) Maximal Number of Subgoals

(b) Maximal Number of Lines per Rule

(c) Indentation (Community Packages)

(d) Indentation (SWI batteries included)

Figure 4: Properties per File.

The length of clauses should be limited as well, both in lines and sub-goals. Fig. 4a and Fig. 4b show the distribution of the largest clause per file. Most clauses are rather short, as many are facts or dispatch to another predicate. There are a few outliers ranging over more than a hundred lines and sub-goals, though most Prolog programmers seem to adhere to a sane limitation.

There are also many whitespace rules suggested by COV, including usage of space(s) after commas in argument lists of a goal (cf. COV 2.5). In the community packages, a large fraction rejects this idea, resulting in 168626 missing spacings on 361782 (46.6 %) lines of code. In the "batteries included" packages, there are 24762 instances on 137051 (18.1 %) of missing whitespace. This suggests that there might be a coding guideline for the SWI-Prolog packages.

We can also find minor differences in superfluous whitespaces at the end of a line (2.4 % vs. 1.1 %), missing linebreaks after the rule operator ":-" (6.0 % vs. 1.0 %), and missing newlines after a sub-goal (6.9 % vs. 0.4%) when comparing the community (first number) and the "batteries included" (second number) packages. However, there are a few instances of a missing newline after a clause in the community packages (0.1 %), while there are none in "batteries included".

4.2 SWI-Specific Features

SWI-Prolog includes many syntactic elements not part of ISO Prolog. Thus, we are interested in how far they have been adopted. In Table 2, the amount of usages of several features is given, again partitioned

Table 2: Adaptation of SWI-Specific Features in the Prolog Community.

		Community Packages		Batteries Included	
Feature	Example	Occurr.	in _ of 251 packs	Occurr.	in _ of 34 packs
shebang	`#!swipl`	11	6 (2.4 %)	2	2 (5.9 %)
digit groups	`1_000`	77	11 (4.4 %)	4	2 (5.9 %)
dicts	`a{b:1}`	883	35 (13.9 %)	119	10 (24.4 %)
unicode character escape	`\u2C6F`	13	2 (0.8 %)	50	7 (20.6 %)
missing closing "\"	`\x00`	9	3 (1.2 %)	0	0 (0.0 %)
single quote char constant	`0''`	13	5 (2.0 %)	1	1 (2.9 %)
zero arguments compound	`pi()`	60	8 (3.2 %)	0	0 (0.0 %)
tab in quotes	`"\t"`	159	9 (3.6 %)	0	0 (0.0 %)
integer exp. notation	`1e3`	38	8 (3.2 %)	16	3 (8.8 %)

into community packages and "batteries included" packages.

As can be seen, most features are not very widespread yet. One reason is that some syntactic elements are very situational, e.g., usage of tabulators in quotes, whereas others have suitable alternatives, e.g., the additional exponential notation. Additionally, these features were introduced fairly recent while many packages already exist significantly longer. Thus, it is quite unlikely that existing code is re-written in order to use new language features.

Yet, dicts in SWI-Prolog seem to be an exception: around 14 % of community packages and 25 % of those shipped with SWI-Prolog make use of dicts at some point. This is an indicator of both active development and interest in the community to use proper associative data structures.

4.3 Limitations of Analysis

Although to the best of our knowledge, our tool *library(plammar)* is compliant to the ISO Prolog standard, it has the following known limitations:

- SWI-Prolog and others offers full Unicode support. Although we handle character escapes like `\u2C6F` for "∀", we do not yet support the literal appearance of "∀" as part of a comment, atom, etc.
- SWI-Prolog allows "[]" and "{}" to be defined as block operators [27, Sec. 5.3.3]. In addition, *quasi quotations* have been added in 2003 [28]. Both techniques are often used for embedding domain-specific languages into Prolog, but are not yet supported by *library(plammar)*.
- In contrast to the ISO standard, SWI-Prolog supports nesting bracketed comments.

We are working to remove these limitations in order to reduce the current rate of 10 % of the files that cannot be processed. Besides the missing support for the stated language extensions, there are some general problems that occur on static source code analysis for Prolog:

- *No explicit export of operator definitions.* In general, parsing Prolog requires knowing all operator definitions within a package. Although Part II of the ISO standard on Prolog's module system introduces the directive `module/2`, used to declare a module and all its exported operator definitions, a module's entry points remain unclear. Hence, for our empirical research, we collected all

operator definitions given in the package's files first. Otherwise, one had to analyze the package's call hierarchy. This is impossible in general, as the module system allows accessing (possibly) internal module files.

– *No explicit import of operator definitions.* In addition, some of SWI-Prolog's "batteries included" and community packages introduce new operators. For our empirical research, we therefore created a list of popular packages with their exported operators.

– *No explicit engines.* In our empirical research, we treated all Prolog files as if designed for SWI-Prolog. This is no restriction, because the language extensions introduced in Section 3.4 are backward compatible to the ISO Prolog standard. Nevertheless, it is desirable to make the targeted Prolog system explicit to allow testing for specific language features. For instance, digit groups have been added to SWI-Prolog of version 7, whereas the definition of the back quote "`" as an operator was only allowed up to version 6.

These problems could be addressed by a more powerful package system, as suggested in [25] or [3]. It would help to resolve dependencies including their operator definitions. A simple solution could be to add suchlike properties to a module meta-description.

5 Related Work

The importance of coding guidelines has been widely accepted throughout different programming languages and paradigms. Existing collections of guidelines range from academic papers, such as the ones we mentioned, to whole books, e.g., in case of Java [10]. Even for low-level languages such as specialized assembly codes, coding guidelines have been suggested [19]. In particular, several coding guideline have been suggested with security in mind, e.g., for C [21, 23].

Quite often, adherence to coding guidelines is improved by relying on tools. For Prolog, there is only a small number of existing development tools for linting as well as code reformatting. Logtalk [14], an object-oriented logic programming language that extends and leverages Prolog, ships with several compiler-based checks that can be also applied to standard Prolog. Integrations for Prolog into popular IDEs such as Visual Studio Code[9] make use of SWI-Prolog's introspection capabilities, and SWI-specific predicates for code formatting, e.g., `portray_clause/2`. Prolog implementations of the language server protocol (LSP)[10], which provides a unified API that is supported by most IDEs, follow the same approach[11].

Parsing of Prolog by expanding the EBNF rules given in the ISO Prolog standard by an additional parse tree argument has been suggested by Nogatz et al. (2019). Seipel et al. (2003) analyze and visualize Prolog source code whose parse tree is given in XML. This representation is the base for program refactorings [4]. A more detailed catalogue of Prolog refactorings is presented in [22].

Analyzing Prolog code by only using term expansions as introduced in Section 3 has been the base for semi-automatic refactoring of large Prolog systems. In [13], an expert system consisting of almost 1 million lines of Prolog code was ported to SWI-Prolog after specifying appropriate rewrite rules for compatibility. However, the changes are applied just at compile-time, without any modifications on the original source code. With *library(plammar)*, existing code can be reformatted according to the coding guidelines.

[9] VSC-Prolog: `https://github.com/arthwang/vsc-prolog`

[10] LSP: `https://microsoft.github.io/language-server-protocol/`

[11] E.g., `https://github.com/jamesnvc/lsp_server`, and `https://github.com/LukasLeppich/prolog-vim`

6 Conclusion and Future Work

In summary, we presented a linter for Prolog, that supports the coding guidelines presented by Covington et al. (2012) as far as we deemed feasible. While certain rules could easily be checked automatically, other remain out of reach for now. Our tool *library(plammar)* is published as SWI-Prolog package and on GitHub at `https://github.com/fnogatz/plammar` (MIT License).

In consequence, we see three major directions for future research. First, we want to broaden the scope of language features our linter can be applied to. In particular, we want to add support for linting Constraint Handling Rules (CHR) [2] and *library(clpfd)* [24]. To do so, we have to carefully consider, how our linter interacts with Prolog's term expansion capabilities. Furthermore, the current set of coding guidelines should be extended to take into account CHR and CLP(FD) idioms. For instance, to the best of our knowledge, there is not yet a developer's standard on how to format non-trivial rules in CHR.

Second, we intent to extend the number of rules we can check automatically. As outlined above, language processing technologies might enable checking rules regarding pronunciation. Other technique from NLP or argumentation analysis might help to partially capture the quality of certain comments [8]. Both extensions would greatly improve the user experience and should lead to Prolog code that is easier to understand and debug. So far, we do not know about other linters implementing such techniques.

Finally, coding guidelines are best when they are easy to implement. Therefore, we intent to improve developer tools for integrating *library(plammar)*, e.g., to add support for LSP, and ease its usage in a CI pipeline.

References

[1] Michael A. Covington, Roberto Bagnara, Richard A. O'Keefe, Jan Wielemaker & Simon Price (2012): *Coding guidelines for Prolog*. *Theory and Practice of Logic Programming* 12(6), pp. 889–927, doi:10.1017/s1471068411000391.

[2] Thom Frühwirth (1998): *Theory and practice of constraint handling rules*. *The Journal of Logic Programming* 37(1-3), pp. 95–138, doi:10.1016/s0743-1066(98)10005-5.

[3] Daniel Cabeza Gras & Manuel V. Hermenegildo (2000): *A New Module System for Prolog*. In: *Proceedings Computational Logic*, CL '00, Springer, pp. 131–148, doi:10.1007/3-540-44957-4_9.

[4] Marbod Hopfner, Dietmar Seipel & Joachim Baumeister (2005): *A Prolog Tool for Slicing Source Code*. In: *19th Workshop on (Constraint) Logic Programming*.

[5] ISO/IEC 13211-1 (1995): *Information technology – Programming languages – Prolog – Part 1: General core*. ISO Standard, International Organization for Standardization, doi:10.3403/00595017.

[6] David Jeffery (2002): *Expressive Type Systems for Logic Programming Languages*. Ph.D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne.

[7] Randy M. Kaplan (1991): *A Plea for Readable Pleas for a Readable Prolog Programming Style*. *SIGPLAN Not.* 26(2), pp. 41–50, doi:10.1145/122179.122184.

[8] Ninus Khamis, René Witte & Juergen Rilling (2010): *Automatic Quality Assessment of Source Code Comments: The JavadocMiner*. In: *Natural Language Processing and Information Systems*, Springer, pp. 68–79, doi:10.1007/978-3-642-13881-2_7.

[9] Sebastian Krings & Philipp Körner (2018): *plspec – A Specification Language for Prolog Data*. In: *Proceedings Declare 2017 – Conference on Declarative Programming*, *LNAI* 10997, Springer, doi:10.1007/978-3-030-00801-7_13.

[10] F. Long, D. Mohindra, R.C. Seacord, D.F. Sutherland & D. Svoboda (2013): *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs*. SEI Series in Software Engineering, Pearson Education.

[11] R. McLaughlin (1990): *A Plea for a Readable Prolog Programming Style*. *SIGPLAN Not.* 25(11), pp. 75–79, doi:10.1145/101356.101360.

[12] Walaa Medhat, Ahmed Hassan & Hoda Korashy (2014): *Sentiment analysis algorithms and applications: A survey*. *Ain Shams Engineering Journal* 5(4), pp. 1093–1113, doi:10.1016/j.asej.2014.04.011.

[13] Edison Mera & Jan Wielemaker (2013): *Porting and refactoring Prolog programs: the PROSYN case study*. *Theory and Practice of Logic Programming* 13(4-5-Online-Supplement).

[14] Paulo Jorge Lopes de Moura (2003): *Logtalk: Design of an object-oriented logic programming language*. Ph.D. thesis, Department of Informatics, University of Beira Interior, Portugal.

[15] Alan Mycroft & Richard A O'Keefe (1984): *A polymorphic type system for Prolog*. *Artificial intelligence* 23(3), pp. 295–307, doi:10.1016/0004-3702(84)90017-1.

[16] Falco Nogatz, Dietmar Seipel & Salvador Abreu (2019): *Definite Clause Grammars with Parse Trees: Extension for Prolog*. In: *Proceedings of 8th Symposium on Languages, Applications, Technologies (SLATE)*. Accepted, preprint available at `http://www1.pub.informatik.uni-wuerzburg.de/pub/nogatz/SLATE19_Preprint_Nogatz-dcg4pt.pdf`.

[17] German Puebla, Francisco Bueno & Manuel Hermenegildo (1998): *A Framework for Assertion-based Debugging in Constraint Logic Programming*. In: *Proceedings Workshop on Types for CLP*, pp. 3–15, doi:10.1007/3-540-49481-2_43.

[18] Germán Puebla, Francisco Bueno & Manuel Hermenegildo (2000): *An Assertion Language for Constraint Logic Programs*. In: *Analysis and Visualization Tools for Constraint Programming*, Springer, pp. 23–61, doi:10.1007/10722311_2.

[19] J.W. Rymarczyk (1982): *Coding guidelines for pipelined processors*. *Comput. Archit. News* 2, doi:10.1145/964750.801821.

[20] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker & Bart Demoen (2008): *Towards Typed Prolog*. In: *Logic Programming*, Springer, pp. 693–697, doi:10.1007/978-3-540-89982-2_59.

[21] Robert C. Seacord (2008): *The CERT C Secure Coding Standard*, 1st edition. Addison-Wesley Professional.

[22] Alexander Serebrenik, Tom Schrijvers & Bart Demoen (2008): *Improving Prolog Programs: Refactoring for Prolog*. *Theory and Practice of Logic Programming* 8(2), pp. 201–215, doi:10.1017/s1471068407003134.

[23] Trupti Shiralkar & Brenda Grove (2009): *Guidelines for Secure Coding*.

[24] Markus Triska (2012): *The Finite Domain Constraint Solver of SWI-Prolog*. In: *FLOPS*, *LNCS* 7294, pp. 307–316, doi:10.1007/978-3-642-29822-6_24.

[25] Jan Wielemaker (2012): *SWI-Prolog: history and focus for the future*. 152, ALP.

[26] Jan Wielemaker (2014): *SWI-Prolog version 7 extensions*. In: *Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014*, p. 109.

[27] Jan Wielemaker (2017): *SWI-Prolog Reference Manual 7.6*. Technical Report.

[28] Jan Wielemaker & Michael Hendricks (2013): *Why It's Nice to be Quoted: Quasiquoting for Prolog*. In: *Proc. 23rd Workshop on Logic-based Methods in Programming Environments (WLPE)*.