Coding Guidelines for Prolog

MICHAEL A. COVINGTON
Institute for Artificial Intelligence, The University of Georgia, Athens, Georgia, U.S.A.
(*e-mail:* `mc@uga.edu`)

ROBERTO BAGNARA∗
Department of Mathematics, University of Parma, Italy
BUGSENG srl, Parma, Italy
(*e-mail:* `bagnara@cs.unipr.it`)

RICHARD A. O'KEEFE
Department of Computer Science, University of Otago, Dunedin, New Zealand
(*e-mail:* `ok@cs.otago.ac.nz`)

JAN WIELEMAKER
Department of Computer Science, VU University Amsterdam, The Netherlands
(*e-mail:* `j.wielemaker@cs.vu.nl`)

SIMON PRICE
Intelligent Systems Laboratory, University of Bristol, United Kingdom
(*e-mail:* `simon.price@bristol.ac.uk`)

submitted November 15, 2009; revised November 11, 2010; accepted May 2, 2011

Abstract

Coding standards and good practices are fundamental to a disciplined approach to software projects, whatever programming languages they employ. Prolog programming can benefit from such an approach, perhaps more than programming in other languages. Despite this, no widely accepted standards and practices seem to have emerged up to now. The present paper is a first step towards filling this void: it provides immediate guidelines for code layout, naming conventions, documentation, proper use of Prolog features, program development, debugging and testing. Presented with each guideline is its rationale and, where sensible options exist, illustrations of the relative pros and cons for each alternative. A coding standard should always be selected on a per-project basis, based on a host of issues pertinent to any given programming project; for this reason the paper goes beyond the mere provision of normative guidelines by discussing key factors and important criteria that should be taken into account when deciding on a fully-fledged coding standard for the project.

Keywords: Prolog, style, coding standards, debugging, efficiency

∗ The work of R. Bagnara has been partly supported by PRIN project "AIDA2007 — Abstract Interpretation Design and Applications."

1 Introduction

The purpose of programming languages is to make programs readable by people. Coding standards enhance this function, especially when multiple programmers and a need for maintainability are present, but also even in the small projects of a single programmer (one must, after all, read and debug one's own work). Coding standards for Prolog are particularly needed for several reasons:

Availability. As far as we know, a coherent and reasonably complete set of coding guidelines for Prolog has never been published. Moreover, when we look at the corpus of published Prolog programs, we do not see a de facto standard emerging. The most important reason behind this apparent omission is that the small Prolog community, due to the lack of a comprehensive language standard, is further fragmented into sub-communities centered around individual Prolog systems, none of which has a dominant position (contrast this with the situation of Java and Sun's coding conventions for that language (Sun Microsystems, Inc. 1999) or the precedents set for C by the UNIX source code).

Language. Language features that contribute to the power of the language make it quite easy —especially for the non-expert— to get things wrong, sometimes in ways that are difficult to diagnose. For example, the lack of prescriptive typing contributes to the suitability of Prolog for quick prototyping in some application domains. At the same time, the fact that no redundant type information is available to the development tools makes discipline particularly important. Besides *types*, Prolog developers and maintainers are confronted with *modes*: the same arguments of procedures can be inputs, outputs, or both. This gives Prolog conciseness and elegance through reversible predicates but makes it necessary to keep track of which modes are supposed to be supported (within reasonable computational complexity limits) by which predicates.

Compiler technology. Most Prolog compilers give no warnings except about singleton variables. Even though more advanced tools and development environments exist, a disciplined approach is still (and will always be) the best device available to those programming in standard Prolog (and any other language).

It is important to stress that we are not announcing *the* coding standard for Prolog. This is a paper with five authors and, on some points, more than five opinions. Rather, we address numerous issues that the maker of a full-fledged coding standard will have to confront. In some cases, there are good reasons to prefer one alternative to another. Other decisions are arbitrary, like driving on the left or on the right; a community of programmers can choose any of several ways of doing something as long as they are consistent. The classic style guide is *The Elements of Programming Style* (Kernighan and Plauger 1978). Published in 1978 and using examples in PL/I and Fortran, the majority of its advice is independent of programming language and still much needed. Indeed, some of our own guidelines are not Prolog-specific but are included because no style guide for practitioners would be complete without them.

The amount of standardization needed depends on the scale and duration of the programming project. Just like a novel or a scientific paper, the quality of a computer program can only be judged from the degree to which it satisfies the programmer's objectives, i.e., "it works": for example, we might say that a novel works if it becomes a best-seller, the manual of a device works if it enables the average user to effectively operate the device, a scientific paper works if it is published in a prestigious journal. Computer programs written for different purposes may or may not "work" in different ways. Only for very simple programs written to solve very simple tasks we can say that *the program works if it functions correctly.* For even moderately complex programs a more sensible definition could be: *the program works if its developers and maintainers are able to approach the expected behavior of the program over its intended lifespan.*

In some cases correctness and maintainability over a long period of time are not important. For instance, in rapid prototyping of applications (something for which Prolog has its advantages) we might say that *the program (prototype) works if (despite its errors and limitations) it demonstrates that the approach is feasible.* Still, the prototype will be incomplete until it is finished, and its chances of being finished one day (as opposed to collapsing and being abandoned before being of any use) depend on qualities such as readability, extensibility, and whatever else helps the development team.

In a sense, the text of a program is not very different from an argumentative essay. Like an argumentative essay, a nontrivial program is addressed to an *audience*, which needs to be accurately identified for the argumentation part of the program to be effective. As for any argumentation, different audiences will require and be prepared to share a different set of *premises* (hypotheses and preconditions explicitly mentioned in the program text) and *assumptions* (hypotheses and knowledge that are left implicit but that still are essential for the comprehension of the program). Not to mention that, in programming, the intended audience is also heavily influenced by the tools it is used to.

To summarize, the existence of very different purposes and wildly different audiences are such that a full-fledged coding standard can only be decided upon on a per-project basis. Moreover, different, equally reasonable and effective coding standards can, on specific points, recommend plainly opposite things: a coherent whole matters more than the individual bits.

In this paper, which evolved from (Covington 2002), we introduce a set of coding guidelines that can serve as a starting point for the development of effective coding standards. We highlight the aspects of Prolog program development that deserve particular attention and can benefit from regulation; we illustrate the rationale that is behind each of the proposed guidelines; when alternative guidelines can achieve the desired effect, their relative merits are discussed.

The paper is organized as follows: in Sections 2, 3 and 4 we discuss guidelines concerning code layout, the naming of program entities, and documentation, respectively; Section 5 concerns the effective use of language features; Section 6 deals with the development, debugging and testing of program units and their interfaces; Section 7 concludes.

2 Layout

Do not let your programs be hampered by inconsistent layout. Poor layout is painful to work with, is distracting, and can lead to otherwise perfectly avoidable mistakes. Moreover, most text editors, if properly operated, can provide significant assistance in ensuring layout consistency.

2.1 Indent with spaces instead of tabs.

You cannot normally predict how tabs will be set in the editors and browsers others will use on your code. Moreover, mixing tabs and spaces makes searches and substitutions more problematic. All in all, it is better not to use tabs altogether: almost any editor can be set to substitute spaces for tabs automatically when saving your file.

2.2 Indent consistently.

Code is more readable if it is indented consistently. Even though the choice of the "right" indent size often leads to friction within the development team, a decision has to be made. Most editors default to an indent size of 8 and while such a large indent does make shallowly nested indentation obvious, it uses up excessive horizontal screen real estate and makes even moderately nested indentation hard to read. A small indent size of 2 spaces is popular with some experienced Prolog programmers but can make it hard for less keen eyed readers or for those using variable pitched fonts to interpret the indentation depth. The compromise value of 4 has proven popular and practical in both Prolog and other well known languages.

If in doubt: Use an indent size of 4 spaces.

2.3 Limit the length of source code lines.

Lines longer than 80 characters are almost always difficult to read. While 80-column-wide screens are no longer in widespread use, 80 columns is still the default width for many text editors. Some editors may also display the last column in an inconvenient way, so limiting the line length to 79 or 78 characters is a good idea.

High-quality printed listings can usually accommodate no more than 65 or 70 characters per line, depending on the type size and paper size. For maximum readability you may want to lower this limit down to 55 characters (Covington 1994).

2.4 Limit the length (number of lines) of clauses.

The ideal situation would of course be for each clause to fit onto an ordinary computer screen. So 24 lines is the safest choice but lengths up to 48 can fit on modern computer displays. You can consider this a limit not to be crossed, the exceptions being for predicates doing long but conceptually simple sequences of I/O or graphic operations.

More generally, always consider simplifying long clauses, whether or not they fit onto your screen. Since in Prolog the only way to have a loop is to introduce another predicate,[1] and case analyses often (though not always) deserve their own predicates too, it is no hardship to follow this guideline.

2.5 Be consistent in the use of space around commas.

The simplest approach is to follow each comma with a space, just as in English. This improves readability by increasing the visual separation among different arguments of a term. Moreover, as this is consistent with the typesetting conventions used for many other languages,[2] doing so often avoids a disturbing mismatch between the program text, its comments, and the string literals used in the program.

The simplicity of the above guideline is valuable, but it misses the opportunity to reduce the decoding effort for readers. An alternative is thus to use space in a way that helps in recognizing the various uses of commas.[3] One possibility is to distinguish between:

- comma meaning *and-then*: follow it by one or more spaces, an optional comment and a newline (see also Guideline 2.7);
- comma separating arguments of a goal: follow it by one or more spaces;
- comma separating elements of a data structure: do not follow it by spaces.

Of course, in order to break long lines, extra newlines should be allowed in the second and third case. In addition, it is tempting to use the ability of putting one or more spaces to line stuff up, as in:

```
shorten(person(Name,       Address, Age),
        person(Short_Name, Address, Age)) :-
    ...
```

Similarly, it may seem natural (to one person, on a particular day, and in a particular context) to relax the rules in order to help distinguishing "levels", as in `[[1,2,3], [4,5], [6,7,8,9]]`. Note also that `[alpha, beta, gamma]` looks nice because the terms are longer than in `[1,2,3]`, *but* if you allow that to make a difference, you can no longer rely on spaces to tell you anything you want to know.

This is why layout remains an art rather than a science. There will always be a friction between the rules —which tend to be either too complex or too rigid but allow for uniformity— and taste —which can adapt to particular situations in often nicer ways but is personal, variable and sometimes inconsistent.

[1] There have been several proposals to provide in-line loop constructs. See, for example, "Logical Loops" (Schimpf 2002), "Declarative Loops and List Comprehensions for Prolog - B-Prolog" (`www.probp.com/download/loops.pdf`), or Lambdas in ISO Prolog (`http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/ISO-Hiord.html`). We do not discuss these due to lack of agreement in the community.

[2] Both natural and formal languages, such as Ada 95 (Ausnit-Hood et al. 1997) and Java (Sun Microsystems, Inc. 1999).

[3] It was pointed out long ago by critics of Prolog that different commas mean different things. Of course, Prolog is by no means alone in this: take C as an example.

For small individual projects, following some general guidelines with flexibility and applying taste on a case-by-case basis can give good results. However, for projects that are bigger and/or have several developers it is best to take style decisions on the basis of explicit and precise principles rather than on the basis of what feels good at the time: sticking to precise rules may occasionally lead to some program fragments that do not look very good, but the overall readability of the project as a whole will benefit.

2.6 Begin each clause on a new line and indent all but the first line of each clause.

For example, write:

```
same_length([], []).
same_length([_|L1], [_|L2]) :-
    same_length(L1, L2).
```

2.7 Put each subgoal on a separate line.

Putting each subgoal on a new line greatly enhances readability. So, no matter how short some subgoals may be, it is better to use one line for each of them. For example:

```
ord_union_all(N, Sets0, Union, Sets) :-
    A is N / 2,
    Z is N - A,
    ord_union_all(A, Sets0, X, Sets1),
    ord_union_all(Z, Sets1, Y, Sets),
    ord_union(X, Y, Union).
```

The only exception might be for short sequences of closely related subgoals, such as those involving `write` and `nl`. For instance, one could have

```
    write('CPU time = '), write(T), write(' msec'), nl.
```

Note though that the `format/[1,2,3]` built-in predicates (not yet in the Prolog standard, but provided by quite a few Prolog systems with reasonably compatible implementations) provide a better way to underscore that, conceptually, several output operations constitute a single step.

2.8 Use vertical space consistently to improve readability.

It is natural for vertical distance to reflect the logical distance. For example: begin the next clause of the same predicate on the next line; skip a line before the first clause of the next related predicate; skip two lines before the first clause of the next unrelated predicate. Notice that the unindented clause head is enough to separate clauses within the definition of a single predicate: no lines need be skipped here.

2.9 Comment source files, not just the predicates within them.

In addition to the comments on particular clauses or predicates, a Prolog source file should begin with some sort of standard header with meta-data about the file: name, version, author(s), revision history, copyright, license and other information. Next there should be the module declaration (in case a module system is used) and maybe other declarations. Immediately after the declaration a detailed comment should explain what the file is all about, what the central ideas are, possibly with illustrations of the data structures used.

It is best if this long explanatory comment is readable as plain text with a minimum of distracting punctuation: it is thus better to use plain `/*` and `*/` delimiters with no extra stars.[4] (The `%` comment delimiter is reserved for comments pertaining to individual predicates, as described below.) If a file divides naturally into several sections, each section can have its own such comment.

2.10 Use layout to make comments more readable.

While text editors can provide significant help in properly formatting code, for comments the developer is usually on his or her own. Nonetheless, a little extra care can do much to increase the readability and usefulness of comments. For example, instead of writing:

```
% This predicate classifies C as whitespace (ASCII < 33), alphabetic
% (a-z, A-Z), numeric (0-9), or symbolic (all other characters).
```

write something like this:

```
% This predicate classifies C as:
%   - whitespace (ASCII < 33);
%   - alphabetic (a-z, A-Z);
%   - numeric (0-9); or
%   - symbolic (all other characters).
```

Indented lists like this are much easier to read than lists disguised as paragraphs. It is also much less work to add or remove an item or add comments about an item.

2.11 Avoid comments to the right of the code, unless they are inseparable from the lines on which they appear.

Comments on the right are problematic, since they so easily get out of step with the code when small changes are made to the program. Moreover, as soon as there is more than one end-of-line comment in the same predicate, it looks bad if they do not begin in the same column, and maintaining this alignment when editing the program becomes an extra chore.

If you cannot dispense with comments to the right of the code, at least make

[4] But possibly with dashes at the top and bottom to increase visual separation.

them very short: just a few crucial hints and, if needed, a pointer to the predicate documentation. For example:

```
%% pred(...)
%  ...
%  [Note 1] Long comment about how pred/n depends on subgoal/k...
%  ...

pred(...) :-
    ...
    subgoal(...),        % See [Note 1] above.
    ...
```

Note that it is perfectly legitimate to write single- and multiple-line comments on their own lines, indented like the other lines in a clause body.

2.12 Consider using comments consistently as reminders.

With some discipline you can effectively mark the points that require your attention at a later time. One possibility is to use the `%` comment delimiter to create "tags" that can then easily and reliably retrieved with string search. For example:

```
%TBD: <short hint on what remains to be done>
%FIXME: <what/why this does not work in all cases>
%HACK: <why this is not completely satisfactory>
```

Here `TBD` abbreviates *to be done.* These abbreviations correspond to the `//TODO:` comments familiar to C# and Java programmers in various development environments.

In more extreme cases, you can guarantee that an unfinished section of your program is not used by inserting Prolog code that will not work. In an unfinished predicate, insert a call to something like `tbd('description of what is to be done')` and do not define the predicate `tbd`. Then any attempt to call the predicate will throw an error.

A related tip is to always write a special comment, like `%DEBUG`, at the end of each line added solely for debugging purposes. Indeed, forgetting to delete debugging code is a common mistake. By tagging that code with `%DEBUG` or similar, it becomes easy to search and delete all the temporary debugging goals and clauses.

2.13 Make clauses understandable in isolation.

As far as you can easily do so, enable the reader to understand each clause without having read everything above it. For example, if a test is unnecessary because a cut guarantees its truth, say so in a comment at the appropriate place. For example, write:

```
remove_duplicates([First|Rest], Result) :-
    member(First, Rest),
    !,
    remove_duplicates(Rest, Result).
remove_duplicates([First|Rest], [First|New_Rest]) :-
    % First does not occur in Rest.
    remove_duplicates(Rest, New_Rest).
```

The comment keeps the clause from being misunderstood if the reader has not read the preceding clause.

2.14 Indent an additional level between `repeat` *and the corresponding cut.*

This makes a `repeat` structure look more like a loop. Here is an example:

```
process_queries :-
    repeat,
        read_query(Q),
        handle(Q),
        Q = [quit],
    !,
    write('All done.'), nl.
```

2.15 Decide how you want to break long clause heads and subgoals.

Even if you try to limit the number of arguments of your predicates (see Guideline 5.3), and, or perhaps because, you choose your names wisely (see Guidelines 3.1–3.12), you will occasionally end up with a clause head or a subgoal that is too long to fit on a single line. In this case, two alternatives make sense:

```
long_predicate_name(long_arg1, long_arg2, long_arg3,
                    long_arg4, long_arg5) :-
    ...
```

and

```
long_predicate_name(
    long_arg1, long_arg2, long_arg3,
    long_arg4, long_arg5
) :-
    ...
```

The second alternative may be perceived as less pretty, but it keeps predicate name length from affecting indentation. By contrast, the first alternative forces you to change the indentation of `long_arg4` every time the predicate is renamed (which happens fairly often during development). However, the lengths of the arguments still affect layout. Alternatives include placing each argument on its own line or placing only semantically related groups of arguments on the same line.

2.16 Decide how to format disjunctions and if-then-elses.

Prolog programmers are deeply divided on how to format the semicolon (which means "or") and the if-then-else structure (`Goal1 -> Goal2 ; Goal3`). Above all, note that:

- Heavy use of semicolons and if-then-elses may indicate that the program is not broken up into clauses correctly. The main ways to indicate disjunction and conditionality are to provide multiple clauses and control the success conditions of each clause. Use semicolons to mark small portions of a clause as nondeterministic alternatives. Use if-then-else for deterministic conditionals, as an alternative to cuts.
- A semicolon at the end of a line can easily go unnoticed and result into a serious programming error.
- Programmers do not remember the relative precedence of commas and semicolons; `a,b;c` means `(a,b);c` but this fact is not obvious. Thus, layout and parentheses to indicate precedence are recommended.

All but the simplest disjunctions should be displayed prominently using layouts such as

```
(
    disjunct_1
;
    disjunct_2
;
    disjunct_3
)
```

or the more compact

```
(   disjunct_1
;   disjunct_2
;   disjunct_3
)
```

In both styles, parentheses are always present, the closing parenthesis is exactly below the opening one, and the semicolons stick out in a way that makes misunderstanding much more difficult. If all of this seems too prolix, recall the first bulleted item above.

The second style, besides compactness, has the advantage that it naturally generalizes to if-then-elses as follows:

```
(   test_1 ->
    test_1_is_true
;   test_2 ->
    test_2_is_true_and_test_1_is_false
;   both_are_false
)
```

A variation that lets the '`->`' operators be more visible at the expense of more vertical space usage is:

```
(
    test_1
 ->
    test_1_is_true
 ;
    test_2
 ->
    test_2_is_true_and_test_1_is_false
 ;
    both_are_false
)
```

This can also be written as

```
(test_1 ->
    test_1_is_true
;
    (test_2 ->
        test_2_is_true_and_test_1_is_false
    ;
        both_are_false
    )
)
```

which is more expensive both in terms of horizontal and vertical space, but some consider it more readable. A third alternative is to put `->` and `;` at the beginnings of lines:

```
(test_1
->  test_1_is_true
;   (test_2
    ->  test_2_is_true_and_test_1_is_false
    ;   both_are_false
    )
)
```

A further variation is to put each `->` and `;` on a line by itself. Whichever style you choose, you should work out how it will handle nesting.

2.17 Consider using automated tools to improve readability.

A *pretty-printer* is a program that prints your programs in a neat, readable form. Pretty-printers are sometimes called *code beautifiers*.

To be more precise, we should recall that, historically, a "pretty-printer" was a program that altered the white space in the source code (adding and removing

```
%% sum_list(+Number_List, ?Result)
%  Unifies Result with the sum of the numbers in Number_List;
%  calls error/1 if Number_List is not a list of numbers.

sum_list(Number_List, Result) :-
    sum_list(Number_List, 0, Result).

%  sum_list(+Number_List, +Accumulator, ?Result)

sum_list([], A, A).       % At end: unify with accumulator.
sum_list([H|T], A, R) :-  % Accumulate first and recur.
  number(H),
  !,
  B is A + H,
  sum_list(Rest, B, R).
sum_list(_, _A, _R) :-    % Catch ill-formed arguments.
  error('first arg to sum_list/2 not a list of numbers').
```

Fig. 1. Prolog code to be pretty-printed

spaces and line breaks) in order to achieve consistent indentation. For C, `cb` and GNU's `indent` are programs of this type. Some Lisp and Scheme systems have a `pp` function for printing functions prettily. Pretty-printing of this type can greatly improve code readability.

Automatic (re-)indentation is built into many editors, as is the ability to alter the type style (plain, italic, bold, underlined, etc.) and/or the color of tokens in order to indicate their role in the syntax (such as which characters are part of comments, which are part of strings, and which are other program text). An example is Emacs' *electric font lock*. For a token styler that handles several programming languages, including Prolog, see `m2h` by Richard O'Keefe (`http://www.cs.otago.ac.nz/staffpriv/ok/software.htm`).

Nowadays, the term "pretty-printing" refers mainly to hardcopy and to listings that are embedded in documentation, textbooks, and the like. For example, consider the code in Figure 1. The LaTeX `listings` package, which is highly customizable and included in all LaTeX distributions along with documentation, pretty-prints Prolog code that is surrounded with the lines:

```
\lstset{language=Prolog, frame=lines}
\begin{lstlisting}[caption={Pretty-printing example},label=pp-ex]
...
\end{lstlisting}
```

The result is shown in Figure 2. Note how the characters of proportional-pitch type are spread apart to help maintain horizontal alignment.

An alternative approach to presenting Prolog programs elegantly in LaTeX is `pltex` by Michael Covington (`http://www.ai.uga.edu/mc/pltex.zip`). This experimental program renders the same Prolog example as shown in Figure 3. Note

```
%% sum_list(+Number_List, ?Result)
%  Unifies Result with the sum the numbers in Number_List;
%  calls error/1 if Number_List is not a list of numbers.

sum_list(Number_List, Result) :-
    sum_list(Number_List, 0, Result).

%  sum_list(+Number_List, +Accumulator, ?Result)

sum_list([], A, A).          % At end: unify with accumulator.
sum_list([H|T], A, R) :-    % Accumulate first and recur.
  number(H),
  !,
  B is A + H,
  sum_list(Rest, B, R).
sum_list(_, _A, _R) :-      % Catch ill-formed arguments.
  error('first arg to sum_list/2 not a list of numbers').
```

Fig. 2. Prolog listing formatted by LaTeX `listings` package

```
%% sum_list(+Numbers_List, ?Result)
%  Unifies Result with the sum the numbers in Numbers_List;
%  calls error/1 if Numbers_List is not a list of numbers.

sum_list(Numbers_List, Result) ←
    sum_list(Numbers_List, 0, Result).

% sum_list(+Numbers_List, +Accumulator, ?Result)

sum_list([], A, A).             % At end: unify with accumulator.
sum_list([H|T], A, R) ←        % Accumulate first and recur.
  number(H),
  !,
  B is A + H,
  sum_list(Rest, B, R).
sum_list(_, _A, _R) ←          % Catch ill−formed arguments.
  error('first arg to sum_list/2 not a list of numbers').
```

Fig. 3. Prolog listing formatted by Covington's `pltex`

the use of font styles to indicate syntax, and the replacement of `:-` by ←. The approach is inspired by "Algol style" in early computer programming literature (Covington 1994). Also, the aforementioned `m2h` program can produce LaTeX output, in addition to HTML and other output formats. The documentation of the LaTeX `listings` package gives references to other pretty-printers, several of which are applicable to Prolog. Finally, Logtalk (Moura 2003) includes support for several syntax highlighters for publishing and printing both Logtalk and Prolog code.

> Programmers are lulled into complacency by conventions.
> By every once in a while, by subtly violating convention,
> you force [the maintenance programmer] to read every line
> of your code with a magnifying glass.
> — Roedy Green (Green 2010)

3 Naming Conventions

Choosing the right names is among the most crucial activities undertaken by any software developer, especially when the programming language is Prolog (which is untyped, with parameters that can act as input, output or input/output and so forth). Detailed advice on naming can be found in (Ledgard and Tauer 1987) and (Kernighan and Pike 1999, p. 104 ff.).

3.1 Choose a writing style for multiple-word identifiers.

The basic choice is between underscore-style (`like_this`) and internal capitalization ("intercaps," `LikeThis`). A complicating factor is that Prolog requires variables to begin with an uppercase letter and atoms to begin with a lowercase letter. This tempts some programmers to use "camel case" (`likeThis`), i.e., intercaps with the first letter lowercased, which other programmers find particularly unaesthetic.

Recall that intercaps became popular in programming languages that did not allow underscores within names, such as Pascal and Smalltalk. In suppressing the visual separation between words, one seems to be reverting to the early Middle Ages, when handwritten words were not separated (Thompson 1893, page 65), and thus flouting an important readability tool that is a thousand years old.

Since Prolog allows underscores, and underscores resemble spaces, there is much to be said for using underscores everywhere a word separator is needed, both in atoms and in variable names. The question then remains how to capitalize variables and atoms. The possibilities are to stick with one case, as in `Result_So_Far` and `is_boolean_function`, or to choose the most appropriate case for each individual word in a compound identifier, as in `Result_so_far` and `is_Boolean_function`. The former convention is certainly more appropriate for variable names, with the only possible exception constituted by short suffixes a program might consistently use to make the information flow explicit, as in

```
simplify_expressions([E_in|Es_in], [E_out|Es_out]) :-
    simplify_expression(E_in, E_out),
    simplify_expressions(Es_in, Es_out).
```

An alternative practice, recommended by some, is to use intercaps for variable names and underscores within atoms. In any case, consistency must be ensured: knowing the words used in an an identifier and knowing whether it is an atom or a variable should allow any team member to know how to spell it immediately.

If in doubt: Use underscores to separate words in compound identifiers, i.e., write `is_well_formed`, not `isWellFormed`. Prefer `Result_So_Far` to `Result_so_far`.

3.2 Make all names pronounceable.

A predicate named `stlacie` is going to confuse anyone else reading your program —including yourself at a later date— even though it may seem, at the time of writing, to be a perfectly obvious way to abbreviate "sort the list and count its elements."

If you don't enjoy typing long names, type short ones. Call your predicate `sac` or even `sc` while typing in your program, then do a global search-and-replace to change it to `sort_and_count`.

3.3 Avoid using different names that are likely to be pronounced alike.

If you use `foo`, do not also use `fou` or `fu`. Many people remember pronunciations, not spellings. Accordingly, it must be absolutely obvious how to spell a name when all you remember is its pronunciation.

For the same reason, do not mix up `to`, `two`, and `too`. At one time it was fashionable to abbreviate "to" as "2," thereby saving one character. That is how computer programs got names such as `afm2tfm` (for "AFM-to-TFM," a TEX utility) and DOS's `exe2bin`. However, this practice creates too much confusion. Remembering how to spell words *correctly* is hard enough: do not ask readers of your code to remember your creative misspellings, too. With the exception of the widely used *i18n* (for "internationalization," a usage coined at DEC many years ago) and *L10n* (for "localization"), spellings like `l8tr` (or `l8r`?) and `w1r3d` do not facilitate communication; they just make the reader suspect that you are still in high school.

3.4 Within names, do not express numbers as words.

If you have three predicates for which you have no better names, call them `pred1`, `pred2`, and `pred3` — not `pred_one`, `pred_two`, and `pred_three`. This is yet another stratagem to make spellings more predictable from pronunciations.

Furthermore, exported predicates should not have numeric suffixes unless the number is some sort of code number. For example,

```
unicode_4_0_0(?Code, ?Class)
```

might be reasonable.

3.5 Choose sensible names for auxiliary predicates.

If part of the algorithm for your predicate, which you have named `foo`, needs to be placed in another predicate definition, do not immediately call that predicate `foo_aux`; usually there are better alternatives. For example:

- If the auxiliary predicate and the main predicate have different numbers of arguments, their names can be the same.
- If an auxiliary predicate is there to do a case analysis, you can use `_case` as a better suffix than `_aux`.

- If an auxiliary predicate is a loop, `_loop` may be a good suffix.
- If an auxiliary predicate relies on a higher predicate to take care of resource allocation and disposal, `_unguarded` might be a good suffix, as in:

```
foo(...) :-
    acquire_resources(...),
    call_cleanup(foo_unguarded(...), release_resources(...)).

foo_unguarded(...) :-
    ...
```

When all else seems inappropriate `_aux` can be used. Other arbitrary alternatives are `_1`, `_2`, `_3` and so on or `_x`, `_xx` and so forth, but these give up any opportunity to indicate the function of the auxiliary predicate and so should preferably be accompanied by explanatory comments.

3.6 If a predicate represents a property or relation, its name should be a noun, noun phrase, adjective, prepositional phrase, or indicative verb phrase.

Examples include:

- `sorted_list`, `well_formed_tree`, `parent` (nouns or noun phrases);
- `well_formed`, `ascending` (adjectives);
- `in_tree`, `between_limits` (prepositional phrases);
- `contains_duplicates`, `has_sublists` (indicative verb phrases).

3.7 If a predicate is understood procedurally —that is, its job is to do something, rather than to verify a property— its name should be an imperative verb phrase.

Examples include `remove_duplicates` (i.e., the second-person imperative form is used, not `removes_duplicates`) and `print_contents` (not `prints_contents`).

3.8 Choose predicate names to help show the argument order.

For example, `mother_of(A,B)` is ambiguous; does it mean "A is the mother of B" or "the mother of A is B"? Naming it `mother_child` or `mother_of_child` would eliminate the ambiguity.

In general, the predicate name is the only place in Prolog where you can conveniently give the user hints about the argument order. For example, a specification like `tree_size(+T, ?S)` makes it clear that the tree argument precedes the size argument. This means that sometimes the most salient type name needs to come last. For example, to convert lists to trees, we might use `list_to_tree(+L, ?T)`. Module prefixes, on the other hand, can only be prefixes. If you see a goal of the form `tree:from_list(X, Y)`, can you tell which of the arguments is the tree and which the list? Hence, `tree:tree_from_list(X, Y)` should not be seen as gratuitously redundant.

3.9 Use descriptive names for variables wherever possible, and make them accurate.

For example, do not call a variable `Tree` if it is not a tree. You would be surprised at how often such things are done by programmers who have changed their minds midway through constructing a clause.

More generally, there are two things that can be encoded in the variable name: its type and its role in the predicate. You can name the variable after only one of these properties if the other is obvious from the context. For example, if you have a predicate `graph_nodes(+Graph, -Nodes)`, it is already obvious that the role of the second argument is to represent graph nodes, so you may name it `List`, if it is a list, or `Assoc`, if it is an association list; for maximum clarity you may even name it `Node_List` or `Node_Assoc`.

3.10 Use single letter names only in presence of suitable conventions.

Prolog has no global variables; most variables exist only in a very limited context. For conciseness, some programmers adopt program-wide conventions whereby, for instance, `N` is always the total number of elements to process and `I` is always the number of elements processed so far. If you have a convention that `L` and `U` are always the lower and upper bounds of some range, fine. If you have a convention that `C`, `C0`, `C1`, `C2` are always character codes, `S`, `S0`, `S1`, `S2`, are sequences, and so on, fine; make sure that such conventions are prominently stated in the source file documentation or are absolutely obvious from context. Whenever you have the slightest doubt that the code may be tricky to read, forget about single letter names and give your reader some help.

3.11 Consistently name threaded state variables.

There is a special case for variables that receive an initial state, go through some translations, and come to a final output. It is important to name those consistently. For example, one can use names of the forms `State0` (for the initial state), `State1`, `State2` and so on (for the intermediate states), and `State` (for the final state):

```
foo(..., State0, State) :-
    foo_step(State0, State1),
    foo(..., State1, State).
```

An alternative choice is to use the names `State_in` (for the initial state), `State_tmp`, `State_tmp1`, `State_tmp2` and so forth (for the intermediate states), and `State_out` (for the final state):

```
foo(..., State_in, State_out) :-
    foo_step(State_in, State_tmp),
    foo(..., State_tmp, State_out).
```

3.12 Use a singular noun for the first element of a list and its plural for the remaining elements.

For example, match a list of trees to `[Tree|Trees]`. If you use a single letter for a list element, use that letter with an 's' after it for the remaining elements. For example, match a list of trees to `[T|Ts]`. Only do this if something nearby in the context, such as an adequately detailed comment, makes it obvious what the single letter stands for. A compromise, which may be more readable in some situations, is `[T|Trees]`.

If there are two entities for which you would like to use the same single letter, give them both full names, as in

```
term_translations([], []).
term_translations([Term|Terms], [Translation|Translations]) :-
    term_translation(Term, Translation),
    term_translations(Terms, Translations).
```

Here, using `T` would be confusing.

3.13 Decide whether predicate names should carry the types on which they operate

The reason a decision is necessary is that, when you also have module names, you can end up with redundancies such as `quaternion:quaternion_magnitude(Q, A)`. If you are using a module system, and you are always including the module name, then you need not put the same information in the predicate names.

On the other hand, a number of widely accepted libraries use predicate names that carry the types on which they operate. This is the case, e.g., for the ordered set library (using the `ord_` prefix) and the association lists library (using the `_assoc` suffix). This convention predates the module system and is similar to a long-standing convention in C, also predating other methods of managing the namespace.

Even when modules are used, some people like to import predicates from other modules and use them without module prefixes, just like you use predicates provided by the system. This is the historically preferred approach, because people expected to convert Edinburgh Prolog code that used plain files to code that used modules just by adding `:- use_module` directives. In this style, type names are still useful even if they are identical to the module name.

3.14 Do not blindly import styles from other languages.

For instance, do not blindly follow the Java tradition by calling everything in sight `is_xxx` or `get_yyy`. It is fine to name `is_tree/1` a *checker*, that is, a predicate that will only succeed, without leaving choice-points, on well-formed, structurally complete trees. This will help the user to distinguish it from the *generator* — whether its existence in the program is real or only conceivable— called `tree/1`.

Similarly, stay away from the massive overuse of "get" that stems from the Java world. It is fine to use "get" to signal that information is being obtained in an extra-logical way, such as when I/O or the foreign language interface are involved. But do not use "get" for fetching a property via an ordinary logical relation.

> Much of the skill in writing unmaintainable code
> is the art of naming variables and methods.
> They don't matter at all to the compiler.
> That gives you huge latitude to use them
> to befuddle the maintenance programmer.
> — Roedy Green (Green 2010)

4 Documentation

All the good reasons why documentation is crucial to the success of any software project, independently from the languages used to code it, are valid for Prolog code. In addition, features like multiple modes, dynamic code, meta-programming facilities, and the lack of prescriptive typing make documentation even more important than for other programming languages.

On the other hand, writing good documentation is expensive and this cost may not be fully justified for those projects where Prolog is used for rapid prototyping. Each project should choose a consistent documentation standard based on the expected benefit/cost ratio. In this choice an important role can be played by the available tools: if the code is written with a system that provides good support for writing, checking, formatting and retrieving the documentation, it is quite likely that choosing the documentation standard(s) dictated by that system will be the most economical solution.

In this section, we will review the important points to consider when designing a code documentation standard.

4.1 Begin every predicate (except perhaps auxiliary predicates) with an introductory comment in a well-defined format.

Predicates that can be called from elsewhere in the program —from code written by others, or by yourself on a different occasion— must be documented by an introductory comment. Here is an example:

```
%% remove_duplicates(+List, -Processed_List) is det
%
%  Removes the duplicates in List, giving Processed_List.
%  Elements are considered to match if they can
%  be unified with each other; thus, a partly uninstantiated
%  element may become further instantiated during testing.
%  If several elements match, the last of them is preserved.
```

This comment follows the template in Figure 4. Note the overall layout and the use of `%%` to mark the first line. Arguments are represented by variable names preceded by mode specifiers from one of the systems in Table 1, and optionally

```
%% <name>(<mode><variable>[:<type>], ...) is <determinism>
%
%  <description line 1>
%  <description line 2>
%  ...

<name>(<variable>, ...) :-
    ...
```

Fig. 4. Example documentation template

followed by type specifiers. At the end of the first line is a determinism specifier from Table 2.

Argument mode specifiers. The first of the three systems in Table 1 is preferred because it is the most explicit about flow of information. For example, note that it would be wrong to write

```
compare(?R:order, +T1:term, +T2:term)
```

to document the `compare/3` primitive, because `T1` and `T2` are allowed to be variables, and

```
compare(?R:order, ?T1:term, ?T2:term)
```

would be misleading because we expect that `?` arguments will be unified with something. An annotation in this scheme that captures the sensible uses of `"compare/3"`is

```
compare(?R:order, =T1:term, =T2:term)
```

Argument type specifiers. The type specifiers, if included, can be arbitrary types meaningful to the reader. Informal types such as `atomic`, `integer`, `list` and `string` are fine, but it is recommended that any unconventional interpretations be documented near the start of the file, program or application as appropriate. Completely informal descriptions can only work for simple and/or small projects. In more complex cases a BNF-like notation can be adopted or, better, one can borrow the syntax of some formal type system, such as the ones described in (Fages and Coquery 2001; Hermenegildo 2000; Mycroft and O'Keefe 1984; Jeffery et al. 2000).[5]

[5] It has not escaped our notice that the use of formal types in code documentation immediately suggests the possibility of giving the compiler access to them in order to enable useful diagnostics of all kinds, if not optimized compilation. Even though we believe that adding standard ways of *optionally* specifying prescriptive types and other annotations (e.g., modes, determinism, cost, termination) to Prolog would result in a significantly more powerful language, all this is completely beyond the scope of this paper.

Table 1. *Three systems of argument mode specifiers*

Recommended system:	
`*`	`ground` on entry, i.e., must already be instantiated to a term that contains no uninstantiated variables when the predicate is called. Thought of as *input*.
`+`	`nonvar` on entry, i.e., must already be instantiated to a term that is not an uninstantiated variable (although the term may contain an uninstantiated variable) when the predicate is called. Thought of as *input*.
`=`	Thought of as *input* but may be `var` on entry, i.e., may or may not be instantiated when the predicate is called. This argument should not share variables with any argument that might be further instantiated by this predicate; this one will be left as it was found.
`-`	`var` on entry, i.e., must be uninstantiated when the predicate is called. Thought of as *output*.
`/`	Similar to '`-`': this argument should be an unbound variable that, in addition, is not shared with any other argument.
`>`	Thought of as *output* but might be `nonvar` on entry. Instantiation must not control the behavior of the predicate.
`?`	Not specified, i.e., may or may not be instantiated when the predicate is called. Instantiation may control the behavior of the predicate. Thought of as either *input* or *output* or *both* input and output.
System used in PlDoc (see below):	
`+`	`nonvar` on entry, i.e., must already be instantiated to a term that is not an uninstantiated variable (although the term may contain an uninstantiated variable) when the predicate is called. Thought of as *input*.
`-`	`var` on entry, i.e., must be uninstantiated when the predicate is called. Thought of as *output*.
`?`	Not specified, i.e., may or may not be instantiated upon entry. Thought of as either *input* or *output* or *both* input and output.
`:`	A meta-argument. Implies `+`.
`@`	Not further instantiated. Typically used for type-tests.
`!`	Contains a mutable structure that may be modified using `setarg/3` or `nb_setarg/3`.
Simplest system:	
`+`	`nonvar` on entry (normally), i.e., will normally be instantiated to a term that is not an uninstantiated variable (although the term may contain an uninstantiated variable) when the predicate is called. Thought of as *input*.
`-`	`var` on entry (normally), i.e., will normally be uninstantiated when the predicate is called. Thought of as *output*.
`?`	Not specified, i.e., may or may not be instantiated when the predicate is called. Thought of as either *input* or *output* or *both* input and output.

Predicate determinism specifiers. The degree of determinism of a Prolog predicate is an important part of its behavior and so must always be documented. Doing so forces the author to consciously consider the determinism and ensures that this information is available for other users of the predicate. Furthermore, by specify-

Table 2. *Predicate determinism specifiers for the template of* Figure 4

`det`	Must succeed exactly once and leave no choice-points.
`semidet`	Must either fail or succeed exactly once and leave no choice-points.
`multi`	Must succeed at least once but may leave choice-points on the last success.
`nondet`	May either fail or succeed any number of times and may leave choice-points on last success.

ing determinism in a standardized representation, like the one defined in Table 2, this information is made available to automated test tools. Informally, `det` is used for deterministic transformations (e.g., arithmetic), `semidet` for tests, `nondet` and `multi` for generators.

After the predicate specification comes a clear explanation, in natural language, of what the predicate does, including special cases (in the `remove_duplicates/2` example, lists with uninstantiated elements). Normally, you should write the comment before constructing the predicate definition; if you cannot describe a predicate coherently in natural language, then you are not ready to write it.

The basic template can easily be extended to deal with multiple modes, e.g.,

```
%% age(+Name:atom, -Age:integer) is semidet
%% age(-Name:atom, +Age:integer) is nondet
%
%   ...
```

Similarly, it is often convenient to document tightly related predicates with multiple arities using just one documentation block, as in

```
%% rdf_load(+File)
%% rdf_load(+File, +Options)
%
%  ...

rdf_load(File) :-
    rdf_load(File, []).

rdf_load(File, Options) :-
    ...
```

When a documentation system supported by adequate tools is available, it should be given appropriate consideration. For instance, if you use SWI-Prolog, it may be advantageous to use PlDoc, the SWI-Prolog source-code documentation infrastructure (`http://www.swi-prolog.org/pldoc/package/pldoc.html`). The comment template used by PlDoc is not very different from the one exemplified in this guideline. It uses the second set of argument mode specifiers, and it also provides tags such as `@param`, `@throws`, and `@error` for further description of a predicate's requirements and behavior.

4.2 Use comments to make main predicates visibly distinct from auxiliary predicates.

Auxiliary predicates exist only to continue the definition of another predicate, and are not called from anywhere else. For example, recursive loops and subordinate decision-making procedures are often placed in auxiliary predicates. While users of your code need not know about auxiliary predicates, commenting predicates can provide useful technical documentation for maintenance and reuse.

Begin the first line of comments with `%%` when introducing a predicate called from elsewhere in the program, but `%` when introducing an auxiliary predicate. For example, we can have

```
%% remove_duplicates(+List, -Processed_List) is det
%
%  Removes the duplicates in List, giving Processed_List.
%  ...
```

and

```
%   remove_duplicates_loop(+List, -Processed_List) is det
%
%   Processed_List contains the first occurrence of each element
%   of List, in the same order.
```

Auxiliary predicates in Prolog often correspond to loops in other languages; the meaning of the Prolog predicate is the loop invariant, and really should be stated in a comment.

4.3 Use descriptive argument names in the introductory comment.

The argument names `Index`, `List` and `Elem` in the introductory comment of the predicate `nth0/3` below were chosen to convey its intended usage.

```
%% nth0(?Index, ?List, ?Elem)
%
%  True if Elem is the Index'th element of List.
%  Counting starts at 0.

nth0(Index, List, Elem) :-
```

```
    ...
```

Well-chosen argument names can also hint at the expected type of arguments, which is particularly important if explicit argument types are not specified. For example, it is reasonable to expect the argument `List` to be associated with some representation of a list and, in the absence of any other information, this would most likely be the usual Prolog list representation. Likewise, `Index` would, in the absence of any other information, suggest a numeric argument type.

4.4 Argument names in the clauses should preferably be the same as in the introductory comment.

Where practical, use the same argument names in the clause definitions as used in the introductory comment, like this:

```
%% nth0(?Index, ?List, ?Elem)
%
%  True if Elem is the Index'th element of List.
%  Counting starts at 0.

nth0(Index, List, Elem) :-
    ...
```

However, there will be frequent occasions when this is not practical; in the following example, `List` in the comment corresponds to `[X|_]` in one clause and `[_|T]` in the other:

```
%% member(?Elem, ?List)
%
%  True if Elem is a member of List.

member(X, [X|_]).
member(X, [_|T]) :-
    member(X, T).
```

4.5 If code needs elaborate explanation, consider rewriting it.

Prolog lends itself to simple, logical programs: *one predicate, one idea.* The act of producing documentation is helpful also because it gives you a chance to spot design mistakes: if you use 'and' several times while describing a single predicate, that predicate is doing too many jobs and should be broken up.

> You don't have to actively lie,
> just fail to keep comments as up to date with the code.
> — Roedy Green (Green 2010)

5 Language Idioms

These guidelines are particularly addressed to those who are relatively new to Prolog and accustomed to the practices of other programming languages. In particular, we highlight the importance of *steadfastness* and tail recursion, the dangers of abusing cuts and the database, the choice of the right kind of lists depending on the operations to be efficiently supported,

5.1 Predicates must be steadfast.

Any decent predicate must be "steadfast," i.e., must work correctly if its output variable already happens to be instantiated to the output value (O'Keefe 1990). That is,

```
?- foo(X), X = x.
```

and

```
?- foo(x).
```

must succeed under exactly the same conditions and have the same side effects. Failure to do so is only tolerable for auxiliary predicates whose call patterns are strongly constrained by the main predicates.

5.2 Place arguments in the following order: inputs, intermediate results, and final results.

In any programming language, it can get confusing when a procedure has more than a few parameters. C's `stdio`[6] is famously confusing because it sometimes puts the stream argument first and sometimes last. Programming language designers know of three things that can help:

1. Strong types help the compiler notice when parameters are in the wrong order. Prolog does not have such a type system, although the related languages Gödel (Hill and Lloyd 1994) and Mercury (Somogyi et al. 1995) do, and the DEC-10 Prolog type checker (Mycroft and O'Keefe 1984) has inspired several type systems for Prolog. However, even a type system does not help when two arguments have the same type. What would have been a single "reference parameter" in C++ or "in out" parameter in Ada or Fortran is a before/after pair of parameters in Prolog.
2. Keywords are used in many languages, including Common Lisp, Ada, S, DEC Pascal, Mesa, SML (in a sense), and Smalltalk. As yet no keyword facility for Prolog has become well known, let alone widely accepted. Some people have used "wrappers", writing code like

   ```
   generalised_hanoi(pegs(5), discs(10), solution(S))
   ```

[6] Part of the C standard library providing various standard input and output operations.

Single-constructor single-argument types in SML and Haskell can be used this way, but those compilers have been let into the secret. Prolog systems take these wrappers literally and build run-time structures for them, which hurts performance. The big problem is indexing, which usually does not look inside wrappers. This style cannot be recommended.

3. A consistent convention is the only alternative left.

You can of course try to devise your own convention. The one described here is a simplification of the one presented in (O'Keefe 1990, pp. 12–16), which was constructed to be consistent with the built in predicates of then-current Edinburgh Prologs.

To be useful, an argument order convention must not only be consistent, it must be *memorable*, which suggests that it should be built on some sort of natural metaphor. The one used here is Space-Is-Time. If X is known or needed before Y in time, it is placed before Y in the arguments. If X represents an earlier state and Y represents a later state, then X is earlier in the arguments.

The basic rule then is "inputs before outputs," which is compatible with first-argument indexing. When the computer is choosing a clause, or a human is reading a predicate, some of the arguments are more important than others, typically the argument(s) controlling a recursion. These being needed before the others (which might be context parameters or accumulators) are written before them.

Let us consider a little example. We want to walk over a binary tree, collecting labels that include a given flag. There will be four parameters. We will be using a `Labels0`...`Labels` pair to build the list. There will be the `Flag` we want to look for. And there will be the `Tree`. The `Labels` are an output, so that pair go last. Within that pair, `Labels0` precedes `Labels` in the list, so it precedes `Labels` in the parameters. The `Tree` drives the induction, so it is the input we need first. Hence there is no choice here:

```
items_including(Tree, Flag, Labels) :-
    items_including(Tree, Flag, Labels, []).

items_including(empty, _, Labels, Labels).
items_including(node(Label,Left,Right), Flag, Labels0, Labels) :-
    items_including(Left, Labels0, Labels1),
    (member(Flag, Label) ->
        Labels1 = [Label|Labels2]
    ;
        Labels1 = Labels2
    ),
    items_including(Right, Labels2, Labels).
```

There is a noteworthy and bitterly regrettable exception to the "most discriminating inputs first" rule, and that is input/output commands with a Stream argument. These commands were introduced at Quintus without any internal discussion. A vastly better design would have been something like MIT Scheme's `with-input-from-port` and `with-output-from-port`, making it easy for programmers to write composite input/output commands that could be redirected easily without *any* stream argument threading. It would not only have been easier to use, it would have been more efficient, because Stream arguments have to be checked every time they are used. Indeed, a preliminary input redirection facility of this kind was in the Quintus library when `write(Stream, Term)` was introduced. Why do we mention this here? Because there are two important style lessons:

1. Get ideas from more than one other language. Stream arguments were a direct copy from C. The better alternative is found in Scheme and Ada. Scheme and functional languages are often good sources of ideas for Prolog; C and Java are less likely to have ideas that fit well into a (mostly-)declarative framework.
2. Talk to other people *before* you release.

This raises the question of optional arguments. Sometimes a family of predicates can be thought of as a single predicate with optional arguments. There are two patterns you can use:

- required inputs, optional inputs, outputs;
- inputs, required outputs, optional outputs.

Sometimes you might technically be able to have both optional inputs and optional outputs, relying on some required input to discriminate, but people reading the code will have a hard time. Don't do that. Stream arguments are again an exception to the rule, but the Options in `open` and `write_term` and so on show the norm.

The other major exception to the inputs before outputs pattern is `is/2`. This follows the convention of assignment commands in most programming languages, so should not be hard to remember. Note, however, that when SWI-Prolog lets you call your own predicates in arithmetic expressions, it requires them to report their result via their *last* argument, conforming to the usual pattern.

5.3 Try to limit the number of predicate arguments.

A predicate with too many arguments may be trying to do several conceptually separate jobs at once, or some of the arguments may need to be replaced with data structures containing pieces of information bundled together.

5.4 Use cuts sparingly but precisely.

First think through how to do the computation *without* a cut; then add cuts to save work. For further guidance see (O'Keefe 1990, pp. 88–101). Concerning code layout, make sure cuts do not go unnoticed: if a green cut[7] may be placed on the same line as the previous predicate call, red cuts definitely must be on their own line of code.

[7] A *green cut* is one that only prunes away some computation paths not contributing new solutions. A *red cut* is a cut that is not green.

5.5 Never add a cut to correct an unknown problem.

A common type of Prolog programming error is manifested in a predicate that yields the right result on the first try but goes wrong upon backtracking. Rather than add a cut to eliminate the backtracking, investigate what went wrong with the logic. There is a real risk that if the problem is cured by adding the cut, the cut will be far away from the actual error (even in a different predicate), which will remain present to cause other problems later.

5.6 Work at the beginning of the list.

You can get to the first element of a 1000-element list in one step; getting to the last element requires 1000 steps. But do not let this or any other guideline stop you from doing the computation that you need to do. In some cases you can speed up a computation by building a list in the reverse of the order you first thought of.

5.7 Avoid `append` *where speed is important.*

Generally, `append` is slow because it has to go all the way to the end of the first list before adding a pointer to the second. Use it sparingly in speed-critical code. But don't be silly: do not re-implement `append` with another name just to avoid using the original one.

One good tactic is to use `append` liberally in prototyping, and then change to other methods, such as difference pairs, in the finished implementation.

5.8 Use difference pairs (difference lists) to achieve fast concatenation.

Suppose you want to combine the sequences $\langle a, b, c\rangle$ and $\langle d, e, f\rangle$ to get $\langle a, b, c, d, e, f\rangle$. One way is to append the lists `[a,b,c]` and `[d,e,f]`.

But there's another way. Store the sequences as lists with uninstantiated tails, `[a,b,c|X]` and `[d,e,f|Y]`, and keep copies of `X` and `Y` outside the lists (so you can get to them without stepping through the lists, which is what `append` wastes time doing). Then just instantiate `X` to your second list, `[d,e,f|Y]`. *Voilà*: the first list is now `[a,b,c,d,e,f|Y]`; you have added material at the end by instantiating its tail. Later, by instantiating `Y` to something, you can add even more elements. The applicability of this technique is limited by the fact that difference pairs can only be appended once.

Difference pairs are discussed in, e.g., (Covington et al. 1997) and (O'Keefe 1990).

5.9 Recognize the memory advantages of tail recursion.

Tail recursion is recursion in which the recursive call is the last subgoal of the last clause, leaving no backtrack points. In this circumstance, there is no need to create a new environment frame on the stack: control can be directly transferred to the beginning of the predicate. See, e.g., (Covington et al. 1997, Chapter 4).

The memory saved by tail recursion can be very important when a recursive loop has to go through thousands or millions of cycles, such as processing single characters of a large input file. When the recursion depth is measured in dozens or less, the memory saved by tail recursion is not important. Moreover, not all computations can easily be made tail recursive. Accordingly, do not be afraid of body recursion (non-tail recursion) if you need it.

Ordinarily, in your first version of a program, you should use tail recursion if you can easily do so, but use body recursion if that makes it easier to get the code right. Then change body recursion to tail recursion if you find that there is a substantial advantage to doing so.

5.10 Avoid `asserta/assertz` *and* `retract` *unless you actually need to preserve information through backtracking.*

Although it depends on your compiler, `asserta/assertz` and `retract` are usually very slow. Their purpose is to store information that must survive backtracking. If you are merely passing intermediate results from one step of a computation to the next, use arguments.

If you have a dynamic predicate, write interface predicates for changing it instead of using 'bare' calls to `asserta/assertz` and `retract`, so that your interface predicates can check that the changes are logically correct, maintain mutexes for multiple threads, and so forth.

5.11 Consider doing things in batches rather than one at a time.

This is language-independent advice. The reason for it is that if you have several things to do to one data structure, you can often share the overheads among a group of operations.

Balanced binary trees typically take about twice as much memory as lists, but they are superb for inserting, deleting, and testing for the presence of single elements in $O(\log n)$ worst case time. You can compute the union, intersection, set difference, and so on of sets of size m and n in $O(m \log n)$ time. But using sorted lists, you can do these operations in $O(m + n)$ time (see (O'Keefe 1990)). This is why `setof/3` returns a sorted list of answers.

The "merge" paradigm can be used any time you have collections sorted on some key to be combined in some way. For example, if you represent polynomials as lists of Exponent-Coefficient pairs, the usual algorithm for adding them is a merge.

This makes sorting an important efficiency tool in Prolog. Beware of comparing non-ground terms, however.

5.12 For sorting, use merge sort or a built-in sorting algorithm.

Several Prolog books show how "cute" quicksort is in Prolog, using it as an example of the use of list differences, but fail to point out that merge sort is better for Prolog (and for other languages, such as ML and Haskell).

Merge sort, an $O(n \log n)$ sorting algorithm, generally outperforms quicksort, which is also $O(n \log n)$ in the average case but $O(n^2)$ in a surprisingly common worst case. A Prolog implementation of merge sort is given in (Covington et al. 1997). A bottom-up merge sort that is $O(n)$ for already sorted input is given in (Brady 2005).

Quicksort was invented for an Elliot 405 computer with 512 words of memory (Hoare 1962). The performance comparison in that paper is of a drum-based quicksort with a drum-based merge sort. The chief virtues of quicksort are that it does not require extra workspace and that the inner loop is fast *if* comparisons are exceedingly cheap. Neither advantage exists when sorting linked lists. As Hoare showed, quicksort does significantly more comparisons than merge sort.

Perhaps more interestingly, merge sort gives you the chance to combine the values of key-value pairs on the fly; because quicksort often puts records with equal keys in different partitions, quicksort variants have to wait until the end.

It should be stressed that many Prolog systems have efficient built-in sorting algorithms that are likely to be faster than anything you would have written, if they fit your needs.

5.13 Develop your own ad hoc *run-time type and mode checking system.*

Many problems during development (especially if the program is large and/or there are several developers involved) are caused by passing incorrect arguments. Even if the documentation is there to explain, for each predicate, which arguments are expected on entry and on successful exit, they can be, and all too often they are, overlooked or ignored. Moreover, when a "wrong" argument is passed, erratic behavior can manifest itself far from where the mistake was made (and of course, following Murphy's laws, at the most inconvenient time).

In order to significantly mitigate such problems, do take the time to write your own predicates for checking the legality of arguments on entry to and on exit from your procedures. In the production version, the goals you added for these checks can be compiled away using `goal_expansion/2`.

> Global variables save you from having to specify arguments in [predicates]. Take full advantage of this. Elect one or more of these global variables to specify what kinds of processes to do on the others. Maintenance programmers foolishly assume that [Prolog predicates] will not have side effects.
> — Roedy Green (Green 2010)

6 Development, Debugging, Testing

The fundamental principles of reliable program development and testing are the same for all programming languages, but the details of how to apply these principles in Prolog involve ideas that will be new to those who have only worked with conventional languages. In particular, unification, backtracking, and the concept of success vs. failure all introduce new twists to the art of program debugging.

6.1 Invest appropriate (not excessive) effort in the program; distinguish a prototype from a finished product.

Many Prolog programs are exploratory; when trying to figure out whether something can be done, it is reasonable to do each part of it in the easiest possible way, whether or not it is efficient.

6.2 The most efficient program is the one that **does the right computation,** *not the one with the most tricks.*

Using efficiency tricks, you can sometimes double the speed of a computation. By choosing a better basic algorithm, you can sometimes speed a computation up by a factor of a million.

Do not modify your code just for the sake of efficiency until you are *sure* it is actually doing the right computation. Of course there may be more than one "right" computation, so keep in mind that simplicity often leads to speed.

6.3 When efficiency is critical, make tests.

Don't believe what people tell you (including us); do your own experiments. The built-in Prolog predicate `statistics` will tell you the time and memory used by a computation. Many Prolog compilers provide other predicates for measuring efficiency. If available in your Prolog environment, use the profiler to measure and analyze efficiency.

Use the debugger, especially a graphical one if available, to examine choice points (backtrack points) and see whether they are as you expect them to be. Careful control of choice points can dramatically reduce the memory footprint of a computation.

Results from efficiency tests are not usually portable between different Prolog environments. Prolog environment developers usually try to ensure that performance of a particular construct does not deteriorate in later versions, but sometimes an alternative construct may become much faster. Make experiments in the specific version of the Prolog environment that you eventually intend to run your program in. Where experiments lead you to choose one construct over another, document the reason for the choice by adding comments to the code.

6.4 Conduct experiments by writing separate small programs, not by mangling the main one.

You will often have to experiment to pin down the exact behavior of a poorly-documented built-in routine or to determine which of two computations is more efficient. Do this by writing separate small programs so that you will know exactly what you are experimenting with. Do not risk making erroneous changes in a larger program in which you have invested considerable effort. There is always the risk that if you make experimental changes, you will forget to undo them (or, worse, forget *how* to undo them). Even if a code versioning system[8] makes it easy to roll back experimental changes, *microbenchmarks* are often more indicated to assess the efficiency of alternative solutions. In fact, when the experiment is conducted on a large program, the changes frequently have effects that are harder to understand because there is more code for them to interact with.

6.5 Look out for constructs that are almost always wrong.

These include:

- a cut at the end of the last clause of a predicate (exactly what alternatives is it supposed to eliminate?);[9]
- a `repeat` not followed by a cut (when will it stop repeating?);
- `append` with a one-element list as its first argument (use the element and `|` instead).

6.6 Isolate non-portable code.

Suppose you are writing an SWI-Prolog program that passes some commands to Windows. You might be tempted to put calls such as

```
    ...,
    win_exec(X, normal),
    ...
```

all through it. Resist the temptation. Instead, define a predicate:

```
pass_command_to_windows(X) :-
    win_exec(X, normal).
```

Elsewhere in your program, call `pass_command_to_windows`, not `win_exec`. That way, when you move to another Prolog system in which `win_exec` has a different name or works differently, you will only need to change one line in your program.

6.7 Isolate "magic numbers" and other constants.

If a number occurs more than once in your program, make it the argument of a fact. Instead of writing

```
    X is 3.14159 * Y
```

in one place and

```
    Z is 3.14159 * Q
```

[8] Highly recommended: see Guideline 6.18.

[9] Such a cut can be legitimate, e.g., when a subgoal in the last clause leaves choice points that need to be discarded (even though the use of `once/1` is a better option). In any case, a predicate ending with a cut is so often a programming error that it bears careful scrutiny, and if intentional, should be commented as such (and *still* scrutinized).

somewhere else, define a fact:

```
pi(3.14159).
```

and write

```
    pi(P),
    X is P * Y
```

and so forth.

This example may seem silly because the value of π never changes, but if you encode 3.14159 in only one place, you don't have to worry about mis-typing it elsewhere.

And if a number in your program really *does* change —a number denoting an interest rate, or the maximum size of a file, or something— then it is very important to be able to update the program by changing it in just one place, without having to check for other occurrences of it.

It is common practice to define a predicate `setting/2` to store constants that are likely to be changed — essentially, parameters that are hard-coded into the program, such as:

```
setting(require_login, true).
setting(timeout_milliseconds, 500).
setting(output_directory, '/home/users/student42/project/output').
...
```

6.8 Take the extra minute to **prevent** *errors rather than having to find them later.*

This is a basic rule of programming. Don't take the extra hour — just take the extra *minute*. In particular, think through loops. For example, given the predicate

```
count_up(10) :-
    !.
count_up(X) :-
    write(X),
    Y is X + 1,
    count_up(Y).
```

what is the first number printed when you count up from 1? The last number? What happens if you try to count up from 11, or from 0.5? A minute's careful thought at the right time can save you an hour of debugging on the computer.

6.9 Test code at its boundaries (limits).

With `count_up` above, the boundary is 10, so you should be especially attentive to situations where the argument is near 10 — slightly below 10, or exactly 10, or slightly above it.

6.10 Test that each loop starts correctly, advances correctly, and ends correctly.

In `count_up`:

- What is the first number printed?
- After printing a particular number (such as 5), what gets printed next?
- What is the last number printed?

That's the essence of understanding any loop: you have to know where it starts, how it advances from one value to the next, and where it stops.

6.11 Test every predicate by failing back into it.

It is not enough to try

```
?- count_up(5).
```

and see what prints out. You must also try

```
?- count_up(5), fail.
```

and see what happens when `count_up` is forced to backtrack. This will show you why the first clause of `count_up` has to contain a cut.

6.12 Do not waste time testing for errors that will be caught anyway.

If you type

```
?- count_up(five).
```

the program throws an arithmetic exception. That's probably okay; the important thing is that you should know that it will happen.

Do not burden your Prolog programs by testing the type of every argument of every predicate. Check only for errors that (1) are likely to occur, and (2) can be handled by your program in some useful way. For example, what happens if you execute this query?

```
?- count_up(What).
```

The most common "arguments of the wrong type" are uninstantiated arguments. Beware: they match anything!

The documentation for a predicate should make it clear which types of arguments are expected and which patterns of argument instantiation are supported. Where you think it is likely that incorrect argument types might be supplied, then add a check to protect against this. However, to avoid rechecking the arguments at each level of recursion, consider adding a separate entry-point predicate that checks the arguments before invoking the original predicate. The original predicate will, of course, need to be renamed.

6.13 For anything beyond a proof-of-concept prototype, use testing systematically.

Equip yourself with some infrastructure to conduct systematic regression testing. Run tests frequently, at least whenever substantial code changes are made. Some Prolog systems provide excellent tools for unit testing. For example, SWI-Prolog provides `plunit`,[10] a tool that is meant to be portable to other systems as well.

6.14 In any error situation, make the program either correct the problem or crash (not just fail).

In Prolog, *failing* is not the same as *crashing*. Crashing here means calling the ISO Prolog standard built-in predicate `throw/1`. If the exception is not caught by the program, the program will terminate with an error.

Failing is the same as answering a question "no." "No" can be a truthful and informative answer, and it is not the same as saying, "Your question does not make sense," or "I cannot compute the answer." An example:

```
?- square_root(169, 13).
yes

?- square_root(169, 12).
no          % makes sense because 12 is not the square root of 169

?- square_root(georgia, X).
no          % misleading because the question is ill-formed
```

In the last case the program should complain that `georgia` is not a number.

6.15 Make error messages informative.

Whenever your program outputs an error message, that message must actually explain what is wrong, and, if possible, display the data that caused the problem. It should also indicate where the error was detected (e.g., by giving the predicate name). A good example:

```
mysort/3: 'x(y,z,w)' cannot be sorted because it is not a list
```

A bad example:

```
error: illegal data, wrong type
```

That doesn't say *what* data is the wrong type or where it was found.

Error reports in library files or modules should be related to the exported predicate they originated from, not to some internal predicate which the user doesn't know about. To achieve this, it may be necessary to add error handlers to some exported predicates just so that they can handle internal deeply generated exceptions and map them back to meaningful user-level descriptions.

[10] See `http://www.swi-prolog.org/pldoc/package/plunit.html`.

6.16 Master the Prolog debugger; it is simple, powerful, and portable.

The classic Prolog debugger allows you to step through a program. It works the same way in virtually all compilers and is described in (Clocksin and Mellish 2003; Covington et al. 1997) and other books.

If your Prolog environment supports graphical debugging then learn how to use it. The principles are the same as the standard command line debugger, but much more information can be displayed on-screen at once, and the program state is more readily visible.

6.17 Do not use `write` *for debugging.*

The easiest way to see what a program does is to sprinkle it with `print/1-2` output commands (when supported by your Prolog compiler), so that you can see the values of the variables as the computation proceeds. This is a good tactic in all programming languages.

In Prolog, don't let backtracking confuse you. You might want to put a unique number in each `print/1-2` so that you can tell exactly which one produced any particular line of output.

The advantage of using `print/1-2` instead of `write/1-2` is that the output of `print/1-2` can be tailored and, in particular, appropriately abbreviated for debugging whereas `write/1-2` insists on showing you everything.

When the output from your program is not being displayed or where debug output would be hard to see for some other reason, remember that you can use `print/2` to explicitly direct output to the terminal or to a log file.

If you add tracing print commands to a program until you don't need any more, you will find that you have too many to make sense of the output. What you need is selective tracing. For example, in UNIX the C `syslog()` function has a priority argument with values representing emergency, critical condition, error, warning, notice, information, and debugging. This allows the production of tracing output to be controlled by how important it is. That's often not enough. Part of your program may be suspect, and other parts not, so you might want everything about one topic to be reported while only errors should be reported elsewhere.

The answer is language-independent common practice. Look for a "tracing" or "logging" facility that lets you say "here is a message at this severity level about this topic with this content", where the topics and severities that get reported can be selected at run time. SWI-Prolog, for example, has `print_message/2`, which offers severity level and programmer-defined formatting, and `debug/2`, which offers topic and format-string formatting. If you don't find exactly what you need, write your own logging library.

You should normally leave logging commands in your source code and suppress their output by selecting no topics and only severe levels. Do not assume that their presence is hurting performance until you have measurements that prove it. Use compiler options to remove debugging and logging code. If you are using your own logging library, and your Prolog system supports term expansion or goal expansion, you might be able to rewrite unwanted logging goals to `true`.

6.18 Use a revision control system.

Every serious programmer can benefit from a revision control system (version control system). For the individual developer a revision control system provides an audit trail of changes to a program; a way to retrieve previous versions; and a way to manage different branches of your code. For multiple developers, a revision control system makes collaborative work possible.

The lowest form of revision control system is to make a new backup copy of the *entire* project before every work session. This is, of course, wasteful of disk space. Proper revision control systems are available as local, client-server, and distributed systems. Commonly used packages include CVS, Subversion, Mercurial, and Git. There are numerous free and commercial implementations. Choose one that suits your needs and use it.

As well as your Prolog programs, keep your test cases, documentation, data files and associated scripts under revision control too. For scientific work, repeatability is essential so make sure that you flag the version used in a published work. Many revision control systems have a web interface that is an ideal way to make your programs publicly available.

> Be sure to comment out unused code instead of deleting it and relying on version control to bring it back if necessary. In no way document whether the new code was intended to supplement or completely replace the old code, or whether the old code worked at all, what was wrong with it, why it was replaced, etc. Comment it out with a lead `/*` and trail `*/` rather than a `[%]` on each line. That way it might more easily be mistaken for live code and partially maintained.
>
> — Roedy Green (Green 2010)

7 Conclusion

It is widely acknowledged that the adoption of coding standards is one of the key factors in the success of a project: it makes it possible to cut the software maintenance costs; it enables effective collaboration between developers; it may even prevent implosion of the project (due to poor readability, lack of useful documentation, use of conflicting conventions and interfaces . . . plus of course the friction within the development team all this usually generates).

For a language like Prolog, coding standards are even more important, due to the following factors:

1. powerful (and complex) language features, such as multiple modes, dynamic code, meta-programming facilities;
2. lack of prescriptive typing and other machine-checkable declarations;
3. substantial lack of sophisticated software development tools.

Despite its potential benefits, a coherent and reasonably complete set of coding guidelines for Prolog has, to the best of our knowledge, never been published. Moreover, no *de facto* standard seems to have emerged. This can be partly explained by the fact that, also due to the lack of a comprehensive language standard, the user community is fragmented into sub-communities that are centered around individual Prolog systems.

In this paper we have made a first step towards filling this gap. We have highlighted those aspects of Prolog program development that deserve particular attention and would benefit from a disciplined approach. For each of those we have introduced a set of coding guidelines, illustrating the underlying rationale. Where alternative (or even conflicting) guidelines could achieve the desired effect, their relative merits have been discussed.

In addition to guidelines that are valuable for any Prolog programmer, this paper contains advice that may be obvious to the more expert. This is intentional: as truly expert Prolog programmers are in very short supply (logic programming courses have been dropped in many universities), the availability and adoption of even simple coding guidelines can improve the productivity of many projects.

Of course, a good degree of subjectiveness is unavoidable on these matters. Nonetheless, we have explained the reasons why it is important to pay attention to certain aspects of Prolog and provided examples that may serve as a basis for further elaboration. Such an elaboration is unavoidable: due to differences in project purposes, environments and developer communities, a full-fledged coding standard can only be established on a per-project basis. We believe the present paper provides a useful starting point in this respect.

Acknowledgment The authors gratefully thank all those who provided feedback and suggestions to improve the paper, among which: Abramo Bagnara, Alan Baljeu, Jan Burse, Parker Jones, Günter Kniesel, Andrzej Lewandowski, Naomi Lindenstrauss, José Morales, Paulo Moura, Robert Muetzelfeldt, Ulrich Neumerkel, Philip Smith, Sylvain Soliman, Michael Sperberg-McQueen, Walter Wilson, and the anonymous reviewers. The input provided by Andrzej Lewandowski and Paulo Moura was particularly valuable.

References

Ausnit-Hood, C., Johnson, K. A., Pettit, IV, R. G., and Opdahl, S. B., Eds. 1997. *Ada 95 Quality and Style, Guideline for Professional Programmers.* Lecture Notes in Computer Science, vol. 1344. Springer-Verlag, Berlin and Heidelberg, Germany.

Brady, M. H. 2005. 'Runsort'—An adaptive mergesort for Prolog. Computer Science Technical Report TCD-CS-2005-34, Trinity College Dublin, Computer Science Department, College Green, Dublin 2, Ireland.

Clocksin, W. F. and Mellish, C. S. 2003. *Programming in Prolog*, 5th ed. Springer, Berlin and Heidelberg, Germany.

Covington, M. A. 1994. Computer languages in type. *Journal of Scholarly Publishing 26,* 1, 34–41.

COVINGTON, M. A. 2002. Some coding guidelines for Prolog. Available at `http://www.ai.uga.edu/mc/plcoding.pdf`.

COVINGTON, M. A., NUTE, D., AND VELLINO, A. 1997. *Prolog Programming in Depth.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

FAGES, F. AND COQUERY, E. 2001. Typing constraint logic programs. *Theory and Practice of Logic Programming 1,* 6, 751–777.

GREEN, R. 1996–2010. Unmaintainable code. Available at `http://mindprod.com/jgloss/unmain.html`.

HERMENEGILDO, M. V. 2000. A documentation generator for (C)LP systems. In *Computational Logic: Proceedings of the First International Conference (CL 2000)*, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer-Verlag, London, UK, 1345–1361.

HILL, P. M. AND LLOYD, J. W. 1994. *The Gödel Programming Language.* MIT Press, Cambridge, MA, USA.

HOARE, C. A. R. 1962. Quicksort. *The Computer Journal 5,* 1, 10–16.

JEFFERY, D., HENDERSON, F., AND SOMOGYI, Z. 2000. Type classes in Mercury. In *Proceedings of the 23rd Australasian Computer Science Conference (ACSC 2000)*, J. Edwards, Ed. IEEE Computer Society, Canberra, Australia, 128–135.

KERNIGHAN, B. W. AND PIKE, R. 1999. *The Practice of Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

KERNIGHAN, B. W. AND PLAUGER, P. J. 1978. *The Elements of Programming Style*, 2nd ed. McGraw-Hill, Inc., New York, NY, USA.

LEDGARD, H. AND TAUER, J. 1987. *Professional Software; Vol. 2: Programming Practice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

MOURA, P. 2003. Logtalk — Design of an Object-Oriented Logic Programming Language. Ph.D. thesis, Department of Computer Science, University of Beira Interior, Portugal.

MYCROFT, A. AND O'KEEFE, R. A. 1984. A polymorphic type system for Prolog. *Artificial Intelligence 23,* 3, 295–307.

O'KEEFE, R. A. 1990. *The Craft of Prolog.* MIT Press, Cambridge, MA, USA.

SCHIMPF, J. 2002. Logical loops. In *Proceedings of the 18th International Conference on Logic Programming (ICLP 2002)*, P. J. Stuckey, Ed. Lecture Notes in Computer Science, vol. 2401. Springer, Copenhagen, Denmark, 224–238.

SOMOGYI, Z., HENDERSON, F., AND CONWAY, T. 1995. Mercury: An efficient purely declarative logic programming language. In Proceedings of the 18th Australasian Computer Science Conference (ACSC '95), R. Kotagiri, Ed. *Australian Computer Science Communications 17,* 1, 499–512.

SUN MICROSYSTEMS, INC. 1999. Code conventions for the Java programming language. Available at `http://java.sun.com/docs/codeconv/`.

THOMPSON, E. M. 1893. *Handbook of Greek and Latin Palæography.* Kegan Paul, Trench, Trübner & Co, London, U.K.