Some Coding Guidelines for Prolog

Michael A. Covington
Artificial Intelligence Center, The University of Georgia

Printed December 2, 2002

The following coding standards are inspired by Kernighan and Pike, *The Practice of Programming,* and my own usual practices. This is a DRAFT document; feedback is invited. I want to thank Don Potter for a number of helpful suggestions.

1 Internal documentation

1.1 Begin every predicate (except auxiliary predicates) with an introductory comment in the standard 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 properly documented. Here is an example:

```
% remove_duplicates(+List,-ProcessedList)
%
%   Removes the duplicates in List, giving ProcessedList.
%   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.
```

The comment starts with a *template* consisting of the name of the predicate and its arguments.
Each argument is preceded by:

- `+` to indicate that it should already be instantiated when the predicate is called;
- `-` to indicate that it should normally be uninstantiated; or
- `?` to indicate that it may or may not be instantiated.

Rare indeed is the Prolog predicate that actually requires an argument to be uninstantiated (which is what `-` ought to mean). Normally, `-` means only that the argument is *normally* uninstantiated.

Note that `+`, `-`, and `?` are used only in comments. They are not part of the Prolog language.

After the template comes a clear explanation, in English, of what the predicate does, including special cases (in this case, lists with uninstantiated elements).

Normally, you should write the comment before constructing the predicate definition. If you cannot describe a predicate coherently in English, you are not ready to write it.

Comments of this type are not needed for *auxiliary predicates,* which 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. Users of your code need not know about auxiliary predicates.

1.2 Use descriptive argument names in the introductory comment; they need not be the same as those in the clauses.

In the example just given, the names `List` and `ProcessedList` are used only in the comment. The actual code will probably split the lists up with `|`. You'll see the actual code later in this document.

2 Predicate names

2.1 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`.

2.2 Never use two different names that are likely to be pronounced alike.

If you use `foo`, do not also use `foo_` or `fu`.

People remember pronunciations, not spellings. Accordingly, it must be absolutely obvious how to spell a name when all you remember is its pronunciation.

One regrettable computer program that I once saw used the names `menutwo`, `menutoo`, `menu2`, and (probably by accident) `mneu2`.

2.3 Construct predicate names with lower-case letters, separating words with underscores.

For example, write `is_well_formed`, not `isWellFormed`.

This is part of a scheme for mapping pronunciations to spellings consistently. If you have a predicate name that is pronounced "boo hiss," you should know immediately that you will spell it `boo_hiss`, not `booHiss` or `boohiss`. If it were a variable name, it would be `BooHiss` (see below).

2.4 Do not mix up *to*, *two*, and *too*.

At one time it was fashionable to abbreviate "to" as "2," thereby saving one character. That's 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; now you ask me to remember your creative misspellings too? Spellings like `l8tr` (or `l8r`?) and `w1r3d` do not facilitate communication; they just make the reader suspect that you are still in high school.

2.5 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 100% predictable from pronunciations.

2.6 Identify auxiliary predicates by appending `_aux` or `_x`, `_xx`, and so forth.

If part of the algorithm for your predicate, which you've named `foo`, needs to be placed in another predicate definition, call that predicate `foo_aux` or `foo_x`.

There are many examples of `_aux` in *Prolog Programming in Depth* and *Natural Language Processing for Prolog Programmers*. However, `_x` is more concise and lends itself to making a sequence (`_xx`, `_xxx`, etc. – pronounced "ex," "ex ex," and so on).

2.7 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).

2.8 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` (not `removes_duplicates`) and `print_contents` (not `prints_contents`).

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

If you get into the habit of doing this consistently, it's one less set of details to remember about each individual predicate.

Intermediate results are those that represent a "count so far" or the like. They usually appear only in auxiliary predicates.

2.10 Consider how your arguments map onto ordinary English.

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` would eliminate the ambiguity.

Kernighan and Pike, p. 104 ff., have more good advice about predicate (or function) names and arguments.

3 Variable names

3.1 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.

3.2 Construct variable names with mixed-case letters, using capitalization to set off words.

For example, write `ResultSoFar`, not `Result_so_far`.

The purpose of this rule, like rule 2.3, is to make the spelling of a name obvious from its pronunciation, and to make variables look different from predicate names.

3.3 For variables of purely local significance, use single letters.

Specifically, use:

`I, J, K, L, M, N` for integers;

`L, L1, L2, L3...` for lists;

`C, C1, C2, C3...` for single characters or ASCII codes;

`A, B, C..., X, Y, Z` for arbitrary terms;

`H` and `T` for head and tail of a list (when better names are not conveniently available).

3.4 Use a single letter for the first element of a list and a plural name for the remaining elements.

For example, match a list of trees to `[T|Trees]`. This is one of several strategies to keep names short but significant.
When the elements of the list do not need to be described, other convenient notations are `[Head|Tail]`, `[H|T]`, and `[First|Rest]`.

4 Layout

4.1 Use /* */ only to comment out blocks of code; use % for explanatory comments.

The usual way to comment out part of a program is to put `/*` on a line before it and `*/` on a line after it. This works only if the material that you're commenting out does not contain `*/`.

4.2 Use layout to make comments more readable.

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 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.

4.3 If code needs elaborate explanation, consider rewriting it.

Prolog lends itself to simple, logical programs: *one predicate, one idea.* If you use "and" several times while describing a single predicate, that predicate is doing too many jobs and should be broken up.

4.4 Indent all but the first line of each clause.

For example, write:

```
remove_duplicates([First|Rest],Result) :-
  member(First,Rest),
  !,
  remove_duplicates(Rest,Result).

remove_duplicates([First|Rest],[First|NewRest]) :-
  % first element of List occurs only once in it
  remove_duplicates(Rest,NewRest).
```

4.5 If a test is unnecessary because a cut has guaranteed that it is true, say so in a comment at the appropriate place.

See "first element of List occurs only once in it" in the previous example. The comment keeps the clause from being misunderstood if the preceding clause is modified or removed.

4.6 Indent an additional 2 spaces 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.
```

4.7 Put each subgoal on a separate line, except closely related pairs such as `write` and `nl`.

The preceding clause would be harder to understand if it were written like this:

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

Yet quite a few Prolog programmers write that way.

4.8 Skip a line between clauses. Skip two lines between predicates.

By establishing this practice in advance, you don't have to make a decision every time you come to the end of a clause.

4.9 Keep clauses less than 25 lines if possible.

Epic-length clauses should be simplified.

4.10 Consider redesigning any non-auxiliary predicate that has more than 4 arguments.

Predicates with too many arguments are usually trying to do several conceptually separate jobs at once.

4.11 Consider using a prettyprinter for finished printouts.

A *prettyprinter* is a program that prints your programs in a neat, readable form. I am working on a program called PLTEX that translates Prolog into LATEX, so that when you feed in this:

```
%:sum_list(+ListOfNumbers,?Result)
%.  Sums the numbers in the list, giving Result. Crashes with an
%.  error message if first argument is not a list of numbers.

sum_list([],0).                 % Empty list sums to 0.

sum_list([First|Rest],N) :-     % Add first number to sum of rest.
  number(First),
  !,
  sum_list(Rest,R),
  N is First+R.

sum_list(_,0) :-                % Catch ill-formed arguments.
  errmsg('First arg of sum_list/2 must be a list of numbers.').
```

the computer prints this:

---

sum_list(+*ListOfNumbers*, ?*Result*)

Sums the numbers in the list, giving Result. Crashes with an error message if first argument is not a list of numbers.

sum_list([ ], 0). % Empty list sums to 0.

sum_list([*First*|*Rest*], *N*) ← % Add first number to sum of rest.
  number(*First*),
  !,
  sum_list(*Rest*, *R*),
  N **is** *First* + *R*.

sum_list(_, 0) ← % Catch ill−formed arguments.
  errmsg('First␣arg␣of␣sum_list/2␣must␣be␣a␣list␣of␣numbers.').

---

5 Expressing algorithms

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

Many AI 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's efficient.

5.2 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 1,000,000 or more.

Do not modify your code just for the sake of efficiency until you are *sure* it is actually doing the right computation.

5.3 When efficiency is critical, make tests.

Don't believe what people tell you (including me); 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.

5.4 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).

5.5 Use cuts sparingly but precisely.

First think through how to do the computation *without* a cut; then add cuts to save work.

5.6 Never add a cut to correct an unknown problem.

If you don't know how it works, and you fix it blindly, you still won't know whether it's reliable after you've modified it to hide the problem.

5.7 Avoid the semicolon (;) — make separate clauses instead.

Instead of

```
colorful(X) :-
  color(X,Y),
  ( red(Y) ; green(Y) ).
```

extract a new concept:

```
colorful(X) :-
  color(X,Y),
  bright(Y).

bright(red).
bright(green).
```

Now the concept of "bright color" isn't hidden within a disjunction.

Advanced programmers will recognize situations in which semicolons are, after all, the best solution to a problem. In that case, their use is permissible.

5.8 Use parentheses whenever operator precedence is important; do not assume that people have memorized the precedence table.

Quick, is `x,y;z` equivalent to `x,(y;z)`? Rather than trying to remember, use explicit parentheses.

5.9 When you use ; always use parentheses.

This follows from the previous rule.

5.10 Avoid if-then-else structures.

The Prolog if-then-else structure, such as

```
(a -> b ; c)
```

is pronounced "if `a` then `b` else `c`" and means "Try `a`, and if it succeeds, do `b` and cut; otherwise do `c`." It is a way of placing a Pascal or C if-then-else structure in the middle of a Prolog clause.

In general, this is a rather un-Prolog-like way of thinking. See the discussion in *Prolog Programming in Depth*.

5.11 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?);[1]
- 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).

5.12 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.

5.13 Avoid `append` as far as possible.

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.

But don't be silly. Do not re-implement `append` with another name just to avoid using the original one.

5.14 Use 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've added material at the end by instantiating its tail. Later, by instantiating `Y` to something, you can add even more elements.

Difference lists are discussed in *Prolog Programming in Depth*.

[1] The assiduous student may wish to work out the circumstances in which such a cut can be legitimate; there are some.

5.15 Use tail recursion for recursions of unknown depth.

Tail recursion is recursion in which the recursive call is the last subgoal of the last clause, leaving no backtrack points. In this circumstance, execution can jump to the beginning of the predicate rather than creating another goal on the stack. See Chapter 4 of *Prolog Programming in Depth*.

5.16 Recognize that tail recursion is unimportant when the depth of the recursion is limited to about 50 or less.

Do not sacrifice simplicity and clarity to gain a tiny amount of efficiency in these cases. But do use tail recursive algorithms when they are as clear and simple as their non-tail-recursive counterparts.

5.17 Avoid `assert` and `retract` unless you actually need to preserve information through backtracking.

Although it depends on your compiler, `assert` 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.

5.18 For sorting, use mergesort or a built-in sorting algorithm.

Many Prolog compilers have efficient built-in sorting algorithms.

Mergesort, an $O(n \log n)$ sorting algorithm, is given in *Prolog Programming in Depth*. It generally outperforms Quicksort, which is also $O(n \log n)$.

5.19 Instead of sorted lists, consider using trees.

A binary tree never needs sorting because it is never out of order – there's always room to insert elements where they belong. See *Prolog Programming in Depth*.

6 Reliability

6.1 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.

Don't do that. 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 compiler in which `win_exec` has a different name or works differently, you'll only need to change one line in your program.

6.2 Isolate "magic numbers."

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
```

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. However, 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's 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.

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

This is a basic rule. 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.

6.4 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.

6.5 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.

6.6 Test every predicate by forcing it to backtrack.

It's 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.

6.7 Test predicates by supplying arguments of the wrong types.

For example, if you are defining `append` for yourself, check what happens if some or all of the arguments are not lists.

For that matter, 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!

6.8 Do not waste time testing for errors that will be caught anyhow.

If you type

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

the program crashes with an arithmetic error. 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.

6.9 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.* 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.

6.10 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
```

Here is a bad example:

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

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

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

The Prolog debugger allows you to step through a program. It works the same way in virtually all compilers and is described in *Prolog Programming in Depth.*

6.12 Use `write` for debugging.

The easiest way to see what a program does is to sprinkle it with `write`s, 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 `write` so that you can tell exactly which one produced any particular line of output.

6.13 Mark all temporarily altered lines of code with "`%TEST!!!`".

That way you will remember to change them back.

6.14 Use `write('!!!')` to mark places in the program where work remains to be done.

This will keep you from forgetting them.

— end —