Prolog Coding Standards

The University of Melbourne
School of Computing and Information Systems

COMP90048
Declarative Programming

Section Bonus

Semester 1, 2019

Outline

There is no single standard for Prolog programmers to follow when writing code. There have been attempts, however, to outline a set of guiding principles. These principles cover:

1. Naming conventions for variables and predicates;
2. Layout guidelines including separation and organisation of predicates, clause design, and construct patterns;
3. Documentation, both at predicate and file level.

This guide draws upon the excellent article:

Covington, Michael A., et al. "Coding guidelines for Prolog." Theory and Practice of Logic Programming 12.6 (2012): 889-927.

Naming Conventions: The Good, The Bad, and The Ugly

Variable and predicate names should be descriptive and accurate.
Uninformative names lead to code that is impossible to interpret.

The Ugly

```
boop(Beep, Bop, Blam) :-
    length(Plop, Beep),
    append(Plop, Blam, Bop).
```

Naming Conventions: The Good, The Bad, and The Ugly

Variable and predicate names should be descriptive and accurate.
Uninformative names lead to code that is impossible to interpret.

The Bad

```
drop_N_elements_from_List(NumElements, List1, List2) :-
    length(List3, NumElements),
    append(List3, List2, List1).
```

Naming Conventions: The Good, The Bad, and The Ugly

Variable and predicate names should be descriptive and accurate.
Uninformative names lead to code that is impossible to interpret.

The Good

```
drop(N, List, Back) :-
    length(Front, N),
    append(Front, Back, List).
```

Naming Conventions: Variables

1. Descriptive, accurate, and pronounceable.

   e.g., Front and Back

2. Choose a convention for multiple-word identifiers.

   e.g., most common to use InterCaps for variables, and to separate words with an underscore in atoms

3. Use single letter names only in presence of a suitable convention.

   e.g., N for a number of elements, L for a list, T for a tree

4. Avoid very long variable names.

   e.g., NoOneNeedsAVariableNameAsLongAsThis

Naming Conventions: Predicates

1. Descriptive, accurate, and pronounceable.

   e.g., parent(...) and not pnt(...)

2. Use nouns, noun phrases, verbs, adjectives, and prepositional phrases.

   e.g., parent (noun), puzzle_solution (noun phrase), sort (verb), connected (adjective), between_limits (prepositional phrase)

3. Choose a convention for multiple-word predicate names.

   e.g., most common to separate words with an underscore

4. Do not import conventions from other languages!

   e.g., get_xxx

Organising Code

1. Group related predicates together.
2. Be consistent with use of vertical whitespace, e.g: a single line between related predicates; two lines between non-related predicates; and no lines between clauses of the same predicate.
3. One goal per line, unless you have a short sequence of related goals.

   e.g., a short sequence of write statements
4. Stick to an 80 character limit on length of lines.
5. Covington et al. recommend that clauses be no more than 24 to 48 lines so that each clause is entirely visible on a computer screen. This is quite long, and for this subject you should not need to write clauses that are longer than a dozen or so lines.

Organising Code

Be consistent with tabs, spaces, and indentation.

All but the head of a clause should be indented. Use indentation to manage long lists of arguments. For example,

```
predicate( argument1,
           argument2,
           argument3,
           ...
         )
```

```
predicate(
            argument1,
            argument2,
            ...
)
```

These are only two possible options for breaking argument lists across lines. The key is to adopt a consistent convention.

Organising Code: Disjunctions and If-Then-Else

Be consistent with how you write disjunctions and if-then-else constructs, and how you nest disjunctions. There is no one correct convention.

Conventions differ in where the semicolon (;) and arrow ($\rightarrow$) and located.

For example, here are two possible conventions for formatting disjunctions:

```
(
    goal1
;
    goal2
;
    goal3
)
```

```
(   goal1
;   goal2
;   goal3
)
```

The idea is to make the presence of semicolons prominent.

Organising Code: Disjunctions and If-Then-Else

Be consistent with how you write disjunctions and if-then-else constructs, and how you nest disjunctions. There is no one correct convention.

Conventions differ in where the semicolon (;) and arrow ($\rightarrow$) and located.

There are a number of different ways of formatting if-then-else constructs.

```
(
        test1
    ->
        goal1
    ;
        test2
    ->
        goal2
    ;
        goal3
)
```

```
( test1 ->
    goal1
; test2 ->
    goal2
; goal3
)
```

Organising Code: Disjunctions and If-Then-Else

Be consistent with how you write disjunctions and if-then-else constructs, and how you nest disjunctions. There is no one correct convention.

Conventions differ in where the semicolon (;) and arrow ($\rightarrow$) and located.

There are a number of different ways of formatting if-then-else constructs.

```
( test1 ->
      goal1
;
      (test2 ->
           goal2
      ;
           goal3
      )
)
```

```
( test1
->  goal1
;   ( test2
    ->  goal2
    ;   goal3
    )
)
```

Documenting Code: File Level Documentation

You code files should have two levels of documentation – file level; and predicate level comments.

Your file should start with some documentation outlining:

1. The purpose of the file;
2. The author;
3. The date at which the code was written;
4. A brief (high-level) summary of what the code does & your rationale.

Documenting Code: Predicate Level Documentation

Provide comments above each predicate in a consistent format, identifying: the meaning of each argument; what the predicate does; and the modes in which the predicate is designed to operate.

For this subject, identifying predicate arguments with one of the following modes is sufficient:

- `+` The argument is viewed as an *input* and must be instantiated to a term that is not an unbound variable when the predicate is called.
- `-` The argument is viewed as an *output* and is provided as a variable when the predicate is called.
- `?` The argument is viewed as either an input or an output, and may or may not be an unbound variable when the predicate is called.

Documenting Code: Predicate Level Documentation

An example from Covington et al. (2011):

```
%% 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.
```

Predicate level documentation should give a concise description of what the predicate does, and the strategy it uses.

Documenting Code: Predicate Level Documentation

Commenting within predicates should be used *sparingly*, and only when necessary to explain a particularly important step. Too much commenting within a predicate compromises readability. Comments on the same line as a goal should be avoided (it is preferable to give comments their own line), unless the comment is particularly short.

Do not make vacuous or obvious comments (e.g., compute the result, compute N - 1, or call append). These comments do not add to a reader's understanding of the code.

Lessons from Project 1

- Inadequate or non-existent file level documentation. File level documentation needs to outline the overall strategy that you have followed to solve the problem;
- Uninformative comments attached to lengthy functions;
- Lack of abstraction (repeating logic that could have been abstracted into its own function);
- Inconsistency in naming styles (camelCase and underscore_names);
- Uninformative naming practices;
- Breaking your programs into sections and providing additional section level documentation was viewed positively.