PROLOG PROGRAMMING STANDARDS
COMPUTER SCIENCE 359 – FALL 2012

1. INTRODUCTION

Why do we need programming standards? This document contains Prolog programming standards, but this information from the official Java website is still very much applicable:

> Code conventions are important to programmers for a number of reasons:
>
> - 80% of the lifetime cost of a piece of software goes to maintenance.
> - Hardly any software is maintained for its whole life by the original author.
> - Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
> - If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Further:

- You gain experience using programming standards representative of those in the operational Air Force and other professional environments.
- Since your work will be graded by someone other than yourself, it is imperative that you do all that you can to ensure your code is understandable. If your instructor cannot understand it, it will not be graded as correct.
- You avoid penalties on each programming exercise for not adhering to these standards!

2. SOURCE FILE STRUCTURE

2.1. The elements of your Prolog source files should be arranged in the following order:

a. file header

b. groupings of facts with fact headers

c. groupings of rules with rule headers

3. FILE HEADER

3.1. Each Prolog source code file shall have a descriptive comment at the beginning (file header):

3.1.1. The main file header comment will contain at least the following information:

```
%   File: File Name Here
%
% Author: Your Name Here
%
% Description: Description of contents of file here.
%
% Documentation: Documentation statement here.
```

3.1.2. If your project contains multiple files, each file will have the above information in a header comment except the documentation statement.

4. FACT HEADER

4.1. Facts should be grouped together and preceded by descriptive comments. A template is provided here:

```
% ------------------------------------------------------------------
% fact_name( parameters ).
% Purpose: High-level description of the semantics of the fact.
% Parameters: Description of each parameter.
% Use: Sample use of the fact.
```

4.2. An example is provided here:

```
% ------------------------------------------------------------------
% like( thing1, thing2 ).
% Purpose: Defines the relationship "thing1 likes thing2".
% Parameters: thing1 - the thing that likes another thing,
%             thing2 - the thing that is liked by another thing.
% Use: like( cadet, falcon ).
%      like( comSciMajors, programmingStandardsDocuments ).
```

5. RULE HEADER

5.1. Rules should be grouped together with one high-level descriptive comment for the group of rules and a more detailed descriptive comment preceding individual rules.

5.2. A template for the high level comment for a group of rules is provided here:

```
% ------------------------------------------------------------------
% rule_name( parameters ).
% Consequent: High-level description of the semantics of the rule;
%             i.e., what is implied if this rule is true.
% Parameters: Description of each parameter.
% Use: Sample use of the rule.
```

5.3. A template for the detailed comment for individual rules is provided here:

```
% Antecedent: Detailed description of the semantics of the rule.
rule_name( parameters ) :- antecedent1( parameters ),
                           antecedent2( parameters ).
```

5.4. An example is provided here:

```
% ancestor( A, D ).
% Consequent: A is an ancestor (parent, grandparent, etc.) of D.
% Parameters: Ancestor - The elder person in the family tree.
%             Descendant - The younger person in the family tree.
% Use: ?- ancestor( elizabeth, william ).
%      true.
%      ?- ancestor( elizabeth, D ).
%      D = william ;
%      D = harry ;
%      false.

% Antecedent: A is an ancestor of D if A is a parent of D.
ancestor( A, D ) :- parent( A, D ).

% Antecedent: A is an ancestor of D if there is a person P who is a
%             parent of D and also A is an ancestor of P.
Ancestor( A, D ) :- parent( P, D ), ancestor( A, P ).
```

6. OUTPUT RULE HEADER

6.1. Rules with the intended purpose of generating output should have a high-level descriptive comment.

6.2. A template is provided here:

```
% ---------------------------------------------------------------
% rule_name( parameters ).
% Purpose: High-level description of the output of the rule.
% Parameters: Description of each parameter.
% Use: Sample use of the rule.
```

6.3. An example is provided here:

```
% ---------------------------------------------------------------
% siblings( Person ).
% Purpose: Display a list of siblings of the given Person.
% Parameters: The Person whose siblings are to be displayed.
% Use: ?- siblings( william ).
%      harry.
```

7. FUNCTION AND ARGUMENT NAMES

7.1. Prolog programs historically do not use the same mixture of upper and lower case within an identifier as Java programs. Most identifiers are all lower-case with underscores used as delimiters. Since this is mostly due to historical reasons, we will also use camel-case.

7.2. Fact and rule names ***must*** begin with a lower-case letter (this is a Prolog syntax requirement) and multiple words will be delimited by an underscore or upper-case letters.

7.3. Variable names ***must*** begin with an upper-case letter (this is a Prolog syntax requirement) and multiple words will be delimited by an underscore or upper-case letters.

8. GENERAL

8.1. Indentation

8.1.1. All code will be properly indented to clearly show the logical structure of the code.

8.1.2. The SWI-Prolog environment does an excellent job indenting Prolog code as it is typed. Be sure to follow this indentation style.

8.2. Commenting and Whitespace

8.2.1. Inline comments should be included in your code where necessary.

8.2.2. Particularly complex code segments should have comments. These comments may need to explain both what is being done and why it is being done at this point in the code.

8.2.3. Whitespace (spaces and blank lines) should be used consistently to enhance the readability of your code. Do not write entire rules all on one line unless they are very, very short.

8.2.4. Especially long lines of code should be broken and indented to be easily readable. No single line of code should exceed the width of a standard laptop screen, generally about 100 characters.

8.3. Other Issues

8.3.1. Rules should be logically cohesive. In other words, all code in a given function should work to perform a single, narrowly focused task.

8.3.2. Group related rules together in your source file.