Prolog Commenting

Basics

Prolog supports both multi-line and single-line comments.

Multi-line comments use C-style comment delimiters:

```
1  /* This is
2  a multiline
3  comment. */
```

Note that SWI-Prolog allows `/*...*/` delimiters to be nested inside each other, something that C (and the Prolog ISO standard) do not. You're welcome to write nested multi-line comments in class, however, since we are using SWI-Prolog.

For single-line comments, Prolog uses the percent sign (%) as the comment delimiter:

```
1  % This is a single line comment.
```

Documenting Predicates

For documenting predicates, we will use a Prolog adaptation of Javadoc-style comments. Place the comment for a predicate **above** the first fact or rule for that predicate:

```
1  /**
2   * concat(+List1 : list, ?List2 : list, ?List3 : list).
3   * concat(?List1 : list, ?List2 : list, +List3 : list).
4   *
5   * Succeeds if the third list is the concatenation of the first two.
6   *
7   * @param List1 The first (left) list to join.
8   * @param List2 The second (right) list to join.
9   * @param List3 The list containing all the elements of List1
10  *              followed by all the elements of List2.
11  */
12 concat([], List, List).
13 concat([Head | Tail], List, [Head | Rest]) :-
14     concat(Tail, List, Rest).
```

First, notice that we cannot determine the argument names, argument types, or parameter modes just by looking at the facts and rules defining the predicate, since these are not explicitly declared in Prolog. As a result, the predicate documentation must begin with one a *declaration header* that defines the argument names, types, and modes. This declaration header includes one or more lines that show alternative *instantiation patterns* for a predicate--that is, indicating different ways the predicate may be called with variables or with bound values in different parameter positions. Each line should use the same formal parameter names for the arguments appearing in the predicate. Arguments can also include a colon (:) and type specifier to indicate the expected type of that parameter.

Typically, arguments are usually one of the following:

`+ArgName`
: A *bound* value, which is often considered an *incoming* value ("in" mode) provided by the caller.

`-ArgName`

An *unbound* value, which means an unbound variable would be supplied by the caller to receive an *outgoing* value ("out" mode).

`?ArgName`
: A value that may be either bound or unbound (or even partially bound), which means the caller may supply either a known value or an unbound variable. The argument can be treated as either incoming or outgoing, depending on what the caller has supplied.

Following the declaration header (and separated by a blank line), a normal Javadoc-style description of what the predicate does should be provided in one or more paragraphs. As with Javadoc, **be sure the first sentence in the predicate's description is a good single-sentence summary.** Additional sentences can provide more detail.

Following the description, use Javadoc `@param` tags to define the meaning/interpretation of each argument. Other Javadoc tags can be used as well where needed. Note that the `@return` tag is typically **not** used, since all predicates either succeed or fail, and the description should indicate when this is so.

Adding Test Cases

Test cases can be added to predicate headers as "examples":

```
/**
 * concat(+List1 : list, ?List2 : list, ?List3 : list).
 * concat(?List1 : list, ?List2 : list, +List3 : list).
 *
 * Succeeds if the third list is the concatenation of the first two.
 *
 * @param List1 The first (left) list to join.
 * @param List2 The second (right) list to join.
 * @param List3 The list containing all the elements of List1
 *              followed by all the elements of List2.
 *
 * Examples:
 *
 * ?- concat([a], [a], [a, b, c]).
 * fail
 *
 * ?- concat([a], [b, c], [a, b, c]).
 * true
 *
 * ?- concat([a], [b, c], C).
 * [ C = [a, b, c] ]
 *
 * ?- concat(A, [b, c], [a, b, c]).
 * [ A = [a] ]
 *
 * ?- concat([a], B, [a, b, c]).
 * [ B = [b, c] ]
 *
 * ?- concat(A, B, [a, b, c]).
 * [ A = [],        B = [a, b, c] ],
 * [ A = [a],       B = [b, c]    ],
 * [ A = [a, b],    B = [c]       ],
 * [ A = [a, b, c], B = []        ]
 */
concat([], List, List).
concat([Head | Tail], List, [Head | Rest]) :-
    concat(Tail, List, Rest).
```

In the examples above, you can see that each **test case** (or **example**) starts with a goal introduced by ?-. Goals may be written across multiple lines, as long as continuation lines **are indented further than the ?-** (use spaces instead of tabs!).

The expected results for each goal are written immediately following the goal, and continue up to the next blank line or the next goal. Expected results take one of three forms:

A comma-separated list of one or more solutions
: When variables are used in a goal, the expected results appearing after the goal should contain a list of one or more solutions separated by commas. Each solution is a list of *Variable = Value* pairs giving the value for each unbound variable listed in the Prolog

goal. A separate list of variable/value pairs should be provided for each distinct solution your predicate can generate. Not that neither the order of the solutions in the output section nor the order of the variables within a solution (if the goal contains more than one unbound variable) matter in the comparison.

`true`
: If the goal contains no unbound variables but should still succeed, list `true` as the expected result. An empty output section will be interpreted the same as `true`.

`fail`
: If the goal has no solution, simply list `fail` (or `false`) as the expected output.

As can be seen from the examples, whitespace in expected results (and in goals) can be used freely. You can break single goals over multiple lines, as long as they are indented. The expected result begins on the first line following the end of the goal, and can also spread over multiple lines. The expected result ends on the first blank line, or when the next goal begins, which ever comes first (no indentation on continuation lines required).

Other Comments in Your Code

JavaDoc comments are "public" documentation of the externally accessible features of your modules. Often, you may also wish to include "internal" (that is, private) documentation that is only useful to someone reading the source code directly. Any comment that does not begin with `/**` is treated as private, purely for someone with access to the source code. You are free to use such comments where ever you like to improve the readability of your code, **but ...**

Internal Comments Are the Documentation Technique of Last Resort

Choose all names carefully so that a naïve reader's first interpretation will always be right. Do not choose names that might mislead someone about what a predicate is supposed to do, or what information a variable holds. Choosing poor names or convoluted logic structure and then trying to explain it in lengthy comments does little to improve readability. This is doubly true for predicates, because half the time a reader will see your predicate name where it is called, **not** when they are reading your predicate itself. If it is not immediately clear what the predicate does, that affects the readability of all the code calling this predicate, no matter how many comments you put in the predicate itself.

Strive to write code that is clear and understandable on its own, simply by virtue of the names you have chosen and the structure you use. If you feel you have to add an internal comment to explain something, ask yourself what needs explaining. If you need to explain what a name refers to or how you intend to use it, consider choosing a better name. If you have to explain a complex series of subgoals or some other convoluted structure, ask yourself (or a TA) if there is a better way. Only after considering these alternatives should you add descriptive comments.

Also, remember that redundant comments that add nothing beyond the obvious are worse than no documentation, so don't write them.

For Experts

The Javadoc-style commenting conventions described here (except for examples used as test cases) are a slightly simplified version of PlDoc format, the Prolog version of Javadoc. You can read more about PlDoc on the SWI-Prolog PlDoc Documentation page. The examples/test cases here are inspired by Python-style doctest comments, and are not a standard component of PlDoc, however.