Using SWI Prolog—The Basics
Notes by James W. Gray, III
Last modified by Shing-Chi Cheung: April 3, 1998

This document gives some very basic information on using SWI Prolog at HKUST. You can start up SWI Prolog by typing `'pl'`.

1. **Interacting with Prolog:** A common way of using SWI prolog is to store your program in a file, load it using the command

   ?- ['filename']. or ?- qcompile(filename).

   during an *interactive session*, and then *query* your program interactively. We'll cover each of these steps using a simple prolog program.

2. **A Simple Prolog Program:** Our first prolog program will consist of a list of facts. A fact states something that is (or we assume to be) true. For example, suppose we want to process information about the relationships among some people. We want to state that "Susan is the mother of Kathryn". We can do this with the following fact.

   ```
   mother('Susan', 'Kathryn').
   ```

   Here, `mother` is called a "functor" and `'Susan'` and `'Kathryn'` are called "arguments". A functor looks like a function name in Pascal or C, but it is different in an important way. It is actually the name of a *relation*. Later on, we'll discuss this concept in detail. For now, we'll simply think of `mother('Susan', 'Kathryn').` as stating that "Susan is the mother of Kathryn".

   Our first prolog program will be the following list of facts.

   ```
   mother('Elizabeth', 'Susan').
   mother('Elizabeth', 'James').
   mother('Elizabeth', 'Joyce').
   mother('Susan', 'Kathryn').
   father('Robert', 'Kathryn').
   female('Kathryn').
   male('James').
   ```

   You can type these facts into a file called "`data.pl`".

   Note that facts can have a single argument. For example, in the above, `female('Kathryn')` states that "Kathryn is female". In general, facts can have any number of arguments.

   Facts are quite limited in their expressive power. Later we will describe a much more general and powerful concept called a "rule".

3. **Interactive Sessions:** After starting SWI Prolog (see the sections on **Getting Access**, above), the first thing you will see is

   ```
   Welcome to SWI-Prolog (Version 2.8.2)
   Copyright (C) 1993-1997 University of Amsterdam.  All rights reserved.

   For help, use ?- help(Topic). or ?- apropos(Word).

   1 ?-
   ```

The character sequence "`| ?-`" is SWI Prolog's prompt. At this point, you can load the facts in the file called `data.pl` by typing

```
1 ?- ['data.pl'].
% compiling file /.../data.pl
% data.pl compiled in module user, 0.050 sec 828 bytes

yes
```

Note the following.

- You don't need to include the `.pl` extension on the file name. For more information on file names for prolog programs, or if you have trouble loading your file, see the section on **Loading Rules**, below.
- All prolog commands end with a period (`.`). If you press return without typing the period, you can type it on the next line and press return again. SWI prolog will process the command as normal.
- The lines

  ```
  % compiling file /.../data.pl
  % data.pl compiled in module user, 0.050 sec 828 bytes

  yes
  ```

  are SWI prolog's response. This response indicates that the facts were successfully loaded.

4. **Querying a Prolog Program:** Once the list of facts are loaded, you can type queries, which prolog will answer according to the facts. For example, you can try the following.

```
| ?- mother('Elizabeth', 'Susan').

yes
| ?- father('Robert', 'Kathryn').

yes
| ?- father('James', 'Kathryn').

no
```

In each case, prolog responds with either `yes` or `no`, depending on whether it finds the fact in the list of facts given in your program.

Queries become more interesting when we use variables. For example, you can try the following.

```
| ?- mother(X, 'Susan').

X = 'Elizabeth'
```

Note the following.

- You must use a *capital* letter for the variable `X`. If you use a lower case `x`, SWI Prolog will treat it as an atom (see the section on **Constants** for more on atoms) and you will get a different result. The general rules for variable names are given below in the section on **Variables**.
- You will notice that SWI Prolog responds with `X = 'Elizabeth'` and then waits for you to do something else; it does not give you the prompt again. For now, you can simply hit the return key and SWI Prolog will give you the prompt.

You might think of the above query as asking "who is the mother of Susan?". More precisely, you should think of it as asking "is there an `X` that is the mother of Susan?". SWI Prolog searches for such an `X` and if it finds one, returns the value.

Consider another query.

```
| ?- mother('Elizabeth', Y).

Y = 'Susan';

Y = 'James';

Y = 'Joyce';

no
```

In this case, when prolog responded with `Y = 'Susan'` and waited, we typed a semicolon (`;`). This tells prolog to search for more solutions. In the above, it found two more solutions, `Y = 'James'` and `Y = 'Joyce'` before it failed and reported `no`.

We can now see some of most significant differences between a function (in Pascal or C) and a relation.

(a) A function can be used in one direction, whereas a relation can be used in either direction. For example, if we have a C function that finds the mother of X, e.g.,

```
void mother(char *X) { ... }
```

we cannot use this function in reverse and find all of the children of a given mother. In contrast, relations work equally well in either direction.[1]

(b) A function can have, at most, a single result. That is, for a given argument, a function can return only one value. In mathematical terms, with each possible argument in the domain, a function associates one value in the range. In contrast, a relation can associate any number of values (in one of its domains) with any number of values (in another of its domains).

An even more dramatic example is the following.

```
| ?- mother(X, Y).

X = 'Elizabeth',
Y = 'Susan' ;

X = 'Elizabeth',
```

[1]However, we will see later that for practical reasons, prolog contains some impurities in the form of "relations" that cannot be reversed. For more information, see the section on **Quasi Relations**.

```
Y = 'James' ;

X = 'Elizabeth',
Y = 'Joyce' ;

X = 'Susan',
Y = 'Kathryn' ;

no
```

Here, we used a variable for both arguments. We are asking "do there exist `X` and `Y` such that `X` is the mother of `Y`?" and prolog searches for all such `X` and `Y`.

5. **Constants:** SWI Prolog has three kinds of constants: integers, floating point numbers, and atoms. Integers and floating point numbers have the usual representations. For example, `993` and `-32` are integers; `3.0`, `-44.993`, `7.24E4` (meaning 72400.0), and `7.24E-3` (meaning 0.00724) are floating point numbers.

   For floating point numbers, there is one restriction to keep in mind. That is, there must be at least one digit before and one digit after the decimal point.

   There are two common ways to write an atom.

   (a) Any sequence of alphanumeric characters, including the underbar ("`_`"), beginning with a lower case letter. For example, `x`, `fishy`, and `john_smith` are atoms.

   (b) Any sequence of characters delimited by single quotes. For example, `'x'`, `'X'`, and `'this atom contains blanks'` are all atoms.

6. **Variables:** In SWI Prolog, variables may be written as any sequence of alphanumeric characters, including the underbar ("`_`"), beginning with either a capital letter or "`_`". Note that if you try to use a variable name that begins with a lower case letter, SWI Prolog will treat it as an atom and you may get some strange results.

   A variable consisting of a single underbar, called an "anonymous variable" is treated specially by prolog. Namely, each occurrence of such a variable is treated as distinct. That is, if you use an anonymous variable twice in the same rule or query, it is *not* considered to be the same variable.

   In many situations, a variable is used in a rule or query a single time. Such an occurrence is called a "singleton variable". In such cases, the name of the variable is superfluous and so an anonymous variable can be used. In fact, such usage is considered good programming style.

7. **Relations:** In SWI Prolog, relations are identified by their name and their *arity*, that is, the number of arguments that they take. For example, the above relation is identified as `mother/2`, since it takes two arguments.

   In C++ you can overload a given name and arity. That is, you can have two distinct functions with the same name and the same number of formal parameters, as long as the compiler can distinguish calls to the two based on the *types* of the actual parameters (i.e., the parameters used in calling the function). However, SWI Prolog is for the most part *untyped*. Therefore, there is no way to overload names with the same arity. You can define two relations with the same name and different arity, e.g., `mother/2` and `mother/3`, but there can be only one relation with a given name and arity. That is, a name/arity pair (e.g., `mother/2`) uniquely determines a particular relation.

8. **Facts:** A relation can be thought of as a table with one column for each argument. For example, the `mother/2` relation would have two columns. Further, the table has one row for every combination of arguments that satisfies the relation. For example, if Susan is the mother of Kathryn, then we would want (Susan, Kathryn) to be one row of the relation.[2] In theory, a relation can have an infinite number of rows.

   In Prolog, we can describe relations by explicitly stating some of the rows that make up the relation. These statements are called *facts*. For example, we can store the following facts in a file called `data1.pl`.

   ```
   mother('Susan', 'Kathryn').
   mother('Elizabeth', 'Susan').
   mother('Elizabeth', 'James').
   father('Robert', 'Kathryn').
   female('Kathryn').
   male('James').
   ```

   Note that we can include facts about more than one relation in the same file. In the above file, there are facts about four different relations: `mother/2`, `father/2`, `female/1`, and `male/1`.

   In the following sections, we will see how we can make use of these facts.

9. **Rules:** *Rules* state that a certain relation holds, given that other relations hold. For example, we may want to say that if `X` is the father of `Y`, then `X` is the *parent* of `Y`. We can state this as the following rule.

   ```
   parent(X,Y) :- father(X,Y).
   ```

   Here, `X` and and `Y` are variables and `:-` is prolog's representation of *if*. You can read the above rule as saying "X is Y's parent if X is the father of Y".

   We would also want to state that "X is Y's parent if X is the mother of Y", so we can include a rule for that too.

   We can include multiple conditions in a rule. For example, suppose we want to say that if X is the parent of Y *and* Y is the parent of Z, then X is the *grandparent* of Z. We can write this in prolog as follows.

   ```
   grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
   ```

   Note that facts are actually special cases of "rules". They are relations that always hold; there is no condition after the if (`:-`). Rules such as the above can be included in the file along with the facts described in the previous section.

10. **Loading rules:** The most convenient and general way I know to load prolog rules into SWI Prolog is to put all of your rules in a file. Suppose your file is named "`myprog`". Once you have started up prolog, you can enter the command

    ```
    ['myprog']. or qcompile(myprog).
    ```

    This will load (and compile) all of the rules in that file. If you want to change your program, you can edit it (e.g., using your favorite editor executing in another window) and then reload it by entering the same `compile` command again.

[2] In relational database terminology, (Susan, Kathryn) would be called a *tuple*.

NOTE: The file that contains your prolog rules can be given a simple name that does not contain any punctuation, such as "`rules`". Alternatively, you can give your file a simple name with an extension of "`.pl`", such as "`rules.pl`". (In either case, you would load the file by issuing the command "`qcompile(rules)`".)

11. **Comments:** As you develop larger files of rules, you will want to include comments in your files. In SWI Prolog, you can write comments by enclosing them with `/*` and `*/`, just as in C and C++.

12. **Queries:** Once you have loaded a file of rules, you can interactively type queries to SWI Prolog. A query is essentially a fact containing some variables. For example, suppose we have a file called `data1.pl`, which, as described above, contains rules defining two relations: `father` and `mother`. We can use these rules as follows.

```
| ?- qcompile(data1).
% compiling file .../data1.pl
% data1.pl compiled in module user, 0.100 sec 1,332 bytes

yes
| ?- mother(X, 'Kathryn').

X = 'Susan'
```

Here, we can think of the query. `mother(X, 'Kathryn').` as asking "is there some X such that `mother(`X`, 'Kathryn')` is true?" In this case, SWI Prolog finds such an X and responds by telling you the value that it finds: `X = 'Susan'`.

After printing this response, SWI Prolog does not give you the prompt; it waits for your input. At this point, you have two choices.

- If you hit the return key, SWI Prolog takes that to mean you have found the answer you want; there's no need to continue. Therefore, SWI Prolog, gives you the prompt and allows you to type a new query.
- If you type a semicolon (`;`) and hit return, SWI Prolog searches for another solution; that is, another value for X. In this case, there are none, so SWI Prolog responds with "`no`".

```
| ?- mother(X, 'Kathryn').

X = 'Susan';

no
```

However, consider the following query.

```
| ?- mother('Elizabeth', Y).

Y = 'Susan' ;

Y = 'James' ;

no
```

In this case, SWI Prolog is able to find two solutions, Susan and James, before it fails.

13. **Exiting SWI Prolog:** You can exit by typing ctrl-D (or whatever is the end-of-file character on your system). Alternatively, you can issue the command

```
halt.
```

14. **Aborting Execution of a Query:** If you think that SWI Prolog has gone into an infinite loop (or you want to stop execution for any other reason), you can interrupt the execution by typing ctrl-C. SWI Prolog will respond:

```
Prolog interruption (h for help) ?
```

You can type `h` and get a list of options or type `q` and the execution will abort (and you'll be back to the main Prolog prompt).

15. **Order of Evaluation:** When executing queries, the order in which prolog searches for rules to apply can make a difference in terms of efficiency. In SWI Prolog, the rules are searched in the order they are declared. In the case of a single file of rules, the search proceeds from the top of the file to the bottom.

Also, the order of evaluation of the conditions in a given rule can make a difference. SWI Prolog evaluates them from left to right.

16. **Negation and Failure to Prove:** The SWI Prolog syntax for "failure to prove" is

```
\+ <predicate>
```

(where `<predicate>` is a prolog predicate).

There is a tendency to think of "failure to prove" as negation. That is, you might think of `\+ <predicate>` as saying that `<predicate>` is false. However, there is a difference. "Failure to prove" can mean either that the predicate is false or that there is not enough information to determine whether it is true or false. For example, suppose the following rules describing the relationships of various people are stored in a file called `data1.pl`.

```
father('James', 'Angeline').
father('James', 'Wesley').
mother('Levan', 'Angeline').
```

During a SWI Prolog session, we could try the following.

```
| ?- qcompile(data1).
% compiling file .../data1.pl
% data1.pl compiled in module user, 0.100 sec 1,332 bytes

yes
| ?- mother('Levan', 'Angeline').

yes
| ?- \+ mother('Levan', 'Wesley').

yes
```

From this, we may want to conclude that Levan is not the mother of Wesley. However, the more accurate conclusion is that prolog *could not prove* that Levan is the mother of Wesley. In particular, at a later time we may add another rule to the file `data1.pl`:

```
mother('Levan', 'Wesley').
```

After adding that rule, we will get a different result:

```
| ?- qcompile(data1).
% compiling file .../data1.pl
% data1.pl compiled in module user, 0.100 sec 1,332 bytes

yes
| ?- mother('Levan', 'Angeline').

yes
| ?- \+ mother('Levan', 'Wesley').

no
```

In this case, prolog was able to prove that Levan is the mother of Wesley. Therefore, `\+ mother('Levan', 'Wesley').` returns `no`.

17. **Lists:** In prolog, lists are represented by a standard functor, `./2` (the "dot functor") that is exactly the same as the cons operator (`::`) in ML. Further, prolog represents the empty list as `[]`. In this way, `.(a, [])` represents the list containing the single element `a`; the term `.(a, .(b, []))` represents a list with two elements; and the term `.(a, .(b, .(c, [])))` represents a list with three elements.

    Clearly, this dot notation quickly becomes cumbersome. Therefore, prolog provides some shorthands.

    (a) A list of items can be written as a comma-separated list delimited by `[` and `]`. For example, the above list of three elements can be written as `[a, b, c]`. To see that SWI prolog really treats these as the same, you can try out the following query.

    ```
    | ?- [a,b,c] = .(a, .(b, .(c, []))).

    yes
    ```

    (b) Prolog provides an alternate notation for decomposing a list into its head and tail; that is, `[ H | T ]`. For example, you can try the following queries.

    ```
    | ?- [a,b,c] = [a|[b,c]].

    yes
    | ?- [a,b,c] = [a,b|[c]].

    yes
    ```

    Beware that the argument after the vertical bar (`|`) should be a list. If it isn't, you will get undesirable results. For example,

    ```
    | ?- [a,b] = [a|b].

    no
    ```

18. **Quasi Relations:** Let f be an n-ary relation and let $t_1, t_2, \ldots, t_n$ be n terms containing m uninstantiated variables, $X_1, X_2, \ldots X_m$. Ideally, when prolog evaluates the goal

    $$f(t_1, t_2, \ldots, t_n)$$

    it will determine whether some instantiation of the variables makes $f(t_1, t_2, \ldots, t_n)$ true. If one or more do, prolog will enumerate them.[3]

    By "enumerate them", we mean the following. Prolog will produce one instantiation; if the execution backtracks (due to the user typing a semicolon or due to the failure of another goal), prolog will produce another instantiation; if the execution backtracks again, prolog will produce yet another instantiation. This goes on until the backtracking stops or no more instantiations exist.[4]

    For example, let's consider the abstract goal $X \geq 3$. (Note: by "abstract" we mean that it is *not* legal prolog.) Ideally, prolog would determine that $X = 3$ satisfies the goal and it would produce that answer. If execution backtracked, prolog would produce $X = 4$, $X = 5$, $X = 6$ and so on. However, this is not what happens.

    SWI prolog has a "relation" (called a built-in predicate) corresponding to $\geq$, which is written `>=`. (For other built-in predicates for comparing numbers, see the section on **Arithmetic Comparison**.) For example, you can try the following.

    ```
    \ ?- 3 >= 3.

    yes
    \ ?- 4 >= 3.

    yes
    \ ?- 5 >= 3.

    yes
    ```

    However, here is what happens when we try `X >= 3`.

    ```
    | ?- X >= 3.
    ! Instantiation error in argument 1 of >= /2
    ! goal:  _7178>=3
    ```

    SWI Prolog is reporting that an error has occurred: `X` is not instantiated. (Note that in the above, `X` has been internally represented as `_7178`.) Why doesn't SWI Prolog simply enumerate the values of `X` that make the goal true? The reason is performance. Even if we don't count floating point numbers, there are so many values of `X` that make this goal true (viz, perhaps on the order of 2^{31}) that enumerating them during the evaluation of some query would ruin the response time of the program. Therefore, it is assumed that the programmer does not intend to do this and an error is reported.

    Thus we see that the built-in predicate `>= /2` works as expected when both of its arguments are fully instantiated. However, if one of its arguments is uninstantiated, an error occurs. More generally, if a relation has limitations requiring some or all of its arguments to be instantiated, we will call it a *quasi relation*.

[3] Actually, prolog would enumerate only the most general instances of instantiated variables. Multiple instances of the most general instances would not be enumerated.

[4] For a proper enumeration, any given instantiation must be found in finite (but perhaps unboundable) time.

Often, the use of quasi relations requires special care. Your programs may have to check whether certain arguments are instantiated before using a quasi relation. See the section on **Testing the Instantiation of Variables** for information on how to do this. In the following, we will be careful to note which built-in predicates are quasi relations and therefore require special care.

19. **Testing the Instantiation of Variables:** When using quasi relations or relations with type restrictions or when improving the performance of your own programs, it is often useful to test whether a variable is instantiated or whether it is instantiated with a particular type. SWI Prolog provides built-in predicates for doing this. Here are the most common ones.

    - `integer/1`: If `X` is an integer, `integer(X)` succeeds. If `X` is something else, such as an atom or an uninstantiated variable, then `integer(X)` fails.
    - `atom/1`: `atom(X)` succeeds if and only if `X` is an atom. In particular, if `X` is an uninstantiated variable, `atom(X)` fails.
    - `atomic/1`: `atomic(X)` succeeds if and only if `X` is a constant, that is, an integer, a floating point number, or an atom.
    - `var/1`: `var(X)` succeeds if and only if `X` is an uninstantiated variable.
    - `nonvar/1`: `nonvar(X)` succeeds if and only `X` is not an uninstantiated variable. It may be a term, a constant, or an instantiated variable.

20. **Arithmetic Unification:** Terms such as `2+3` are not automatically evaluated by prolog. On the contrary, they are stored just like other terms; `+/2` is the functor and `2` and `3` are the arguments. For example, consider the following query.

    ```
    | ?- X = 2+3, X = 3+2.

    no
    ```

    This query fails because the arithmetic expressions are *not* evaluated. `X` is successfully unified with `2+3`, but when prolog tries to unify `2+3` and `3+2`, it fails as soon as it attempts to unify `2` and `3`.

    To overcome such problems, prolog provides a built-in predicate `is/2`, which evaluates its right-hand argument and then unifies the result with its left-hand argument. For example:

    ```
    | ?- X is 2+3, X is 3+2.

    X = 5
    ```

    Note that `is/2` does *not* evaluate its left-hand argument. For example:

    ```
    | ?- 5 is 3+2.

    yes
    | ?- 2+3 is 3+2.

    no
    ```

    In the first case, `3+2` (from the right-hand side) is evaluated to produce `5`, which can then be unified with `5` (from the left-hand side). However, in the second case, `5` cannot be unified with the unevaluated 2+3 from the right-hand side. To handle such situations, see the section on **Arithmetic Comparison**.

Warning: is/2 is a quasi relation. Its right-hand argument must be fully instantiated.

Exercise: Here are two examples that mix unification with is/2. Explain the output. In the second example, explain how an instantiation error is avoided.

```
| ?- X is 2+3, X = 2+3.

no
| ?- X = 2+3, Y is X.

X = 2+3,
Y = 5
```

21. **Arithmetic Comparison:** SWI prolog provides a full set of built-in predicates for comparing arithmetic expressions. They are <, =<, >, >=, \= (not equal), and == (equal). All of these predicates evaluate both of their arguments before performing the comparison.

    Warning: all of these predicates are quasi relations. Both of their arguments must be fully instantiated.

    To compare other kinds of data, see the section on *Non-Arithmetic Comparison.*

22. **Character Lists:**

    A *character list* (or alternatively, a *string*) is written as a sequence of characters delimited by double quotes (e.g., "this is a list"). Internally, a character list is stored as a list of integers—the ASCII codes for the characters. Note that character lists can be manipulated using the normal operations for lists. For example, here's how SWI Prolog responds to the query "test" = [H|X].

    ```
    | ?- "test" = [H|X].

    H = 116,
    X = [101,115,116]
    ```

    This response can be understood from the fact that the ASCII code for "t" is 116, the ASCII code for "e" is 101, etc..

23. **Conversion between Character Codes and Atoms:**

    Since a string such as "test" is stored internally as a list of character codes, it is sometimes useful to be able to do the following conversions.

    - Converting between a single character code and a single-character atom. This can be accomplished using the char_atom/2 predicate. The relation char_atom(X,Y) holds if and only if X is the character code for Y. Here are some examples.

      ```
      | ?- char_atom(X, a).

      X = 97

      | ?- char_atom(X, '3').

      X = 51
      ```

```
| ?- char_atom(51, Y).

Y = '3'

| ?- char_atom(-1, X).

no
| ?- char_atom(X, ab).

no
| ?- char_atom(X, Y).

X = 1,
Y = '';

X = 2,
Y = ''
```

NOTE: the `char_atom` predicate is defined in the `strings` library. Before using it, you must load that library. To see how, see the section on Loading Libraries, below.

- Converting between a character list and a multi-character atom. This can be accomplished using the built-in predicate, `atom_chars/2`. The goal `atom_chars(X,Y)` holds if and only if `Y` is the list of character codes for `Y`.

  Warning: `atom_chars/2` is a quasi relation; at least one of its arguments must be fully instantiated.

  Here are some examples.

```
| ?- atom_chars(ab, X).

X = [97,98]
| ?- atom_chars(X, [97,55]).

X = a7
| ?- atom_chars(X, Y).

! Instantiation error in argument 2 of atom_chars/2
! goal:  atom_chars(_7234,_7235)
```

24. **Non-Arithmetic Comparison:** We can compare variables, numbers, atoms, and compound terms using the built-in predicates "`@<`", "`@=<`", "`@>`", "`@>=`", "`\==`" (not equal), and "`==`" (equal).

    In particular, since strings are character lists, we can compare them. Here are some examples.

```
| ?- "dog" @< "cat".

no
| ?- "dog" @> "cat".

yes
```

When the character lists are not the same length, these relations behave as if the shorter operand is padded on the right with "-1"s. For example:

```
| ?- "fish" @< "fishery".

yes
```

We can think of this response being generated as follows. First, the character list `"fish"` is represented as the list `[102,105,115,104]` and `fishery` is represented as `[102,105,115,104,101,114,12`
Then, the first list is padded with "-1"s to make it the same length as the other; that is, `fish` becomes `[102,105,115,104,-1,-1,-1]`. Now the lists are compared from left to right. In the above case, we find that -1 is less than 101, so the query succeeds and SWI Prolog returns `yes`.

We can also compare atoms using the same relations. In this case, the relations behave as if the operands are first converted to character lists.

Here's an example.

```
| ?- fish @< fishery.

yes
```

In addition, SWI Prolog allows us to compare terms of different types. When comparisons are made, the following rules are used.

- Every uninstantiated variable is less than any number. For example:

  ```
  | ?- X @< 7.

  X = _6740

  | ?- X @> 7.

  no
  ```

- Every number is less than any atom.
- Every atom is less than any compound term. In particular, since strings are character lists and hence compound terms (using the "`.`" functor), every atom is less than any string. For example:

  ```
  | ?- "fish" @< fishery.

  no
  | ?- predator @< "fish".

  yes
  ```

The comparison operators are transitive. For example, since every number is less than any atom and every atom is less than any compound term, we have that every number is less than any compound term.

There are two other points to keep in mind about the non-arithmetic comparisons.

(a) The non-arithmetic comparison operators do *not* evaluate their arguments prior to performing the comparison. This can lead to some surprising results. For example,

```
| ?- 2+5 @< 3+6.

yes
| ?- 5+2 @< 3+6.

no
```

In both cases, both terms are unevaluated. In the first case, `2` is less than `3` and `5` is less than `6`, so the query succeeds. In the second case, `5` is not less than `3`, so the query fails.

(b) When comparing two uninstantiated variables, SWI Prolog uses the internally generated name to determine which is greater. This can lead to some surprising results. For example,

```
| ?- X @> Y.

no
| ?- Y @< X.

Y = _7397,
X = _7414
```

Even though the two queries seem to be asking the same question, they have different results. In the first case, the query fails since the internally generated name used for `X` (e.g., `_7397`) is not greater than the internally generated name used for `Y` (e.g., `_7414`). In the second query, the names are assigned in the reverse order, so the query succeeds.

25. **Loading Libraries** You can load a set of library predicates using the `ensure_loaded/1` predicate. For example, to load the `strings` library, type the following command.

```
| ?- ensure_loaded(library(strings)).
% loading file ...
%  foreign file ... loaded
%  loading file ...
%  types.qof loaded in module strings, 0.050 sec 12,192 bytes
% strings.qof loaded, 0.534 sec 37,452 bytes
% module strings imported into user

yes
```