Put the "Pro" in Prolog

1 Overview

The remainder of the class will focus on the use and implementation of Prolog. Prolog, short for "**Pro**gramming in **log**ic", is a *logical language*, in contrast to *object-oriented languages* like Java and *functional languages* like Scala. True to the namesake, the typical way of thinking in logical languages is based on what logically must be true. This thinking approach tends to emphasize *what* correct solutions are, as opposed to *how* to compute a correct solution.

The purpose of using Prolog in this class is to expose you to yet another way of thinking when you're writing code. While it's unlikely that you will use Prolog outside of this class, the ideas transfer to other domains.

1.1 Motivation

Prolog is particularly well-suited to problems which involve *constraint satisfaction*, or *search*. In these sort of problems, we know *what* a correct solution looks like, but we may not know exactly *how* to produce it. This is classically true for NP-Complete problems, though this applies to a surprisingly wide context. For example, in my own dissertation (which perhaps biases me to use Prolog in this class), I use Prolog to generate test cases for a large variety of software, both industrial and academic in nature. At its core, the idea is to write a very simple implementation of the system being tested in Prolog. Prolog can then search through this implementation and generate test inputs which will hit different parts of the implementation. To be clear, I don't know exactly how to generate good tests, but I do know that good tests should hit a bunch of parts of the implementation. Prolog fills in the gap here to generate such tests.

1.2 Preliminaries

For the rest of this document, we assume you are using SWI-PL (`http://www.swi-prolog.org/`) as your *engine*, where the engine is what is used to execute Prolog code. SWI-PL can be loaded with the `swipl` command on CSIL, which will bring you to the REPL. The prompt for the REPL is `?-`; any code you see in this document which is preceeded by `?-` is intended to be written at the REPL. The code after `?-` is known as a *query*, which serves as a code entry point.

Code in this document which is **not** preceeded by `?-` is intended to be loaded in *before* queries are executed. This code can be loaded in by putting it all into one file (in this case, `foo.pl`), and then issuing the following query:

```
?- [foo].
```

Note that input files **must** have the `.pl` extension.

2 Facts

One of the most basic things we can do in Prolog is define *facts*: statements which are trivially true. For example, consider the following code:

```
1 isInteger(0).
2 isInteger(1).
3 isInteger(2).
4
5 isName(alice).
6 isName(bob).
```

The way to read the above code on a per-line basis is the following:

- 1: `isInteger` is true underneath input `0`
- 2: `isInteger` is true underneath input `1`

- 3: `isInteger` is true underneath input `2`
- 5: `isName` is true underneath input `alice`
- 6: `isName` is true underneath input `bob`

We can issue queries on the above facts like so (where lines which do not begin with `?-` are the output of the engine).

```
1  ?- isInteger(0).
2  true.
3  ?- isInteger(3).
4  false.
5  ?- isName(alice).
6  true.
7  ?- isName(bob).
8  true.
9  ?- isName(carol).
10 false.
```

As shown above, while it is `true` that `isInteger` holds under input `0` in line 1, it is `false` that `isInteger` holds under input `3` in line 3. This follows from the definition of the `isInteger` facts in the previous listing. Similar behavior is seen for the `isName` facts.

2.1 Data Used: Integers and Atoms

There are two kinds of data used in the previous listings: integers and atoms. Integers are, well, integers, and are represented as we would expected (e.g., `0`, `1`, `2`, etc.). Atoms are names which begin with a lowercase letter, as with `alice`, `bob`, and `carol` in the above listings. For our purposes, atoms behave like strings, with the constraint that they **must** begin with a lowercase letter.

> ***Sidenote:*** Strictly speaking, atoms are *symbols*, not strings. The biggest difference between symbols and strings is that multiple syntactic uses of the same symbol will all end up pointing to the same underlying data structure in the engine, whereas multiple syntactic uses of a string may point to different (but identical) data structures in the engine. This makes it cheap to see if two symbols are identical, as this means they would have reference equality. For strings, we would need to look at the contents of the strings, which is much more expensive. However, strings are more amenable to being created on the fly, whereas the symbols used by a program usually remain fixed.

3 Variables

So far, we have only seen how to ask if a fact holds under some known data. In and of itself, this is not particularly useful. However, we can use the same infrastructure to ask a closely related question: under what data does a fact hold?

In order to ask this sort of question, we need to introduce *variables*. Variables **must** begin with an uppercase letter, which distinguishes them from the atoms defined in the previous section. To see variables in action, consider the following code:

```
1  isOne(1).
2
3  isTwo(2).
```

As before, we can still check to see if a fact holds under some known input:

```
1  ?- isOne(1).
2  true.
3  ?- isOne(2).
4  false.
```

However, with variables in play, we can now ask for inputs under which the fact holds, like so:

```
1  ?- isOne(X).
2  X = 1.
3  ?- isTwo(Y).
4  Y = 2.
```

Variables can also be used in the very definition of facts. For example, consider the following code:

```
1  areEqual(X, X).
```

Intuitively, the meaning of the above code is that the `areEqual` fact holds if both of its inputs have the same value. When used with known data, this behaves in a straightforward manner:

```
1  ?- areEqual(1, 1).
2  true.
3  ?- areEqual(1, 2).
4  false.
5  ?- areEqual(2, 2).
6  true.
```

However, with variables in play, the `areEqual` fact begins to show some interesting behavior. Consider the following queries:

```
1  ?- areEqual(X, 1).
2  X = 1.
3  ?- areEqual(1, X).
4  X = 1.
5  ?- areEqual(X, Y).
6  X = Y.
```

All three queries above *succeeded*, meaning they were true. However, because the values of some variables were determined in the process, the engine no longer prints `true`; the `true` is implicit in the fact that some variables received values. At line 1, the engine figured out that in order to make the `areEqual` fact hold underneath the variable `X` and the integer `1`, it must be the case that `X = 1`. This was similarly true at line 3, where the only difference is the order of the parameters to `areEqual`. The query on line 5 is arguably the most interesting of these, because the engine was able to deduce `X = Y`; that is, the variables `X` and `Y` must be equal to each other. The engine deduced this without knowing exactly what these variables are; while we know that `X = Y`, it could be the case that `X = 1` or `X = 2`, but we do not know for certain.

Let's go deeper. Consider the code below, which behaves a lot like `areEqual` above but it operates over three inputs instead of two:

```
tripleEqual(X, X, X).
```

Now consider the following queries:

```
1  ?- tripleEqual(1, 1, 2).
2  false.
3  ?- tripleEqual(X, 1, 2).
4  false.
5  ?- tripleEqual(1, 1, X).
6  X = 1.
7  ?- tripleEqual(X, Y, 1).
8  X = Y, Y = 1.
9  ?- tripleEqual(X, 1, Y).
10 X = Y, Y = 1.
11 ?- tripleEqual(X, Y, Z).
12 X = Y, Y = Z.
```

As one might expect, the query on line 1 does not succeed (it gives back `false`), because `1` and `2` are not equal to each other. The same reasoning applies for the query on line 3; the value of variable `X` is irrelevant, because `1` will never equal `2`. The query on line 5 succeeds with `X = 1`, as all other inputs have been fixed at `1`. The query on line 7 shows that all the variables involved are equal to each other (namely, `X = Y`), but additionally one of those variables equals a value (namely, `Y = 1`). While the engine does not explicitly show it, it transitively holds that `X = 1` for this query, as `X = Y` and `Y = 1`. The exact same reasoning follows for the query on line 9, which differs from line 7 only in parameter ordering. With the query on line 11, all variables are equal to each other (again, while the engine does not explicitly show it, it transitively holds that `X = Z`, as `X = Y` and `Y = Z`).

4 Nondeterminism

An astute reader may have noticed that in the previous section (Section 3), we did not use the same initial example introduced in Section 2. For the purposes of the current section, we will reuse the initial example from Section 2, and use it along with the variables introduced in Section 3. For convenience, this code has been duplicated below:

```
1 isInteger(0).
2 isInteger(1).
3 isInteger(2).
4
5 isName(alice).
6 isName(bob).
```

Consider the following query on the above code, which uses variables:

```
?- isInteger(X).
X = 0
```

The engine appears to hang at this point, as it is waiting for user input. If we hit semicolon (`;`), we can see additional output:

```
?- isInteger(X).
X = 0 ;
X = 1
```

...at which point the engine appears to hang again. After another press of semicolon (`;`):

```
?- isInteger(X).
X = 0 ;
X = 1 ;
X = 2.
```

...at which point the prompt returns and the engine waits for another query.

This sort of behavior may at first appear very strange, as this illustrates a fundamental feature of logic programming languages which is not present in other types of languages. This feature is that of *nondeterministic execution*. What this means is that execution can split-off into effectively different worlds at well-defined points. In the query above, when executing `isInteger(X)`, there are three possibilities based on the `isInteger` facts. As such, execution splits into three different worlds, where each world executes a different `isInteger` fact. A description of each world follows:

- In the first world, the fact `isInteger(0)` is chosen, so `X = 0`.
- In the second world, the fact `isInteger(1)` is chosen, so `X = 1`
- In the third world, the fact `isInteger(2)` is chosen, so `X = 2`

Note that while execution split into three worlds, we visited each of these three worlds in a precise order which reflected the order of the rules in the file. Each time we pressed semicolon (`;`), we effectively requested that the engine explore the next world.

Each separate use of `isInteger` in a query (hereinafter called a *call*) splits the world in this way. That is, if multiple calls to `isInteger` are made, then *each* will split execution in this manner. An example follows.

4.1 Conjunction

We can make multiple calls to `isInteger` via *conjunction*, represented with a comma (`,`). This is also sometimes referred to as a *compound query*. To see this in action, consider the following query:

```
1 ?- isInteger(X), isInteger(Y).
```

Intuitively, because each call to `isInteger` splits the world, we would expect this query to show all possible combinations of `X` and `Y` if we keep hitting semicolon (`;`) for more solutions. While this will happen, these solutions will be in a well-defined order. Because there are so many solutions, we put these in a separate listing below for clarity, in the same order as delivered by the engine:

1. `X = 0, Y = 0`

2. `X = 0, Y = 1`
3. `X = 0, Y = 2`
4. `X = 1, Y = 0`
5. `X = 1, Y = 1`
6. `X = 1, Y = 2`
7. `X = 2, Y = 0`
8. `X = 2, Y = 1`
9. `X = 2, Y = 2`

As shown with the query results above, the world splits on the first call to `isInteger(X)`. In the first world explored, the fact `isInteger(0)` is chosen, and so `X = 0`. Execution then proceeds forward with `X = 0`, until the second call to `isInteger(Y)` is encountered. This call selects the `isInteger(0)` fact to use, so `Y = 0`. At this point, computation has ended for this set of worlds, leading to the overall result `X = 0, Y = 0`.

However, there are still other worlds to explore. The engine will explore different worlds based on the most recent world choice made. In this case, because the most recent choice of worlds was done with `isInteger(Y)`, we choose a different world for this call. The next world has the fact `isInteger(1)`, so `Y = 1`. Nothing is left to execute, so overall `X = 0, Y = 1`. Upon asking for another solution, there are still worlds to explore for the call `isInteger(Y)`, and the fact `isInteger(2)` is chosen, leading to `Y = 2`. Since there is nothing left to compute after this point, overall `X = 0, Y = 2`.

At this point, there are no further choices for `isInteger(Y)`. As such, when asked for another solution, the engine looks for the next most recent choice made, which was done for `isInteger(X)`. Initially, the fact `isInteger(0)` was used for the call to `isInteger(X)`, but at this point we have completey exhausted all the possibilities of that world. As such, the next choice for `isInteger(X)` is made, leading to the usage of the fact `isInteger(1)`, leading to `X = 1`. From here, the call to `isInteger(Y)` is performed, which entails splitting into three worlds corresponding to the three facts for `isInteger`.

Using this same sort of reasoning process, you should be able to reason through the rest of the results on your own. You should **not** proceed until you understand why the rest of the results are received in their shown order; we only build on this later on, so it will likely get more confusing if you do not understand it already.

In order to implement this idea of going back to the most recent choice, we can employ a stack. Note that this stack maintains *choices*: other worlds which we have yet to explore, with the next world to explore on top. This is **completely unrelated** to the call stack; they are completely different data structures.

4.2 Revisiting Facts

The original explanation of facts in Section 2 intentionally didn't mention nondeterminism in order to simplify the discussion. However, even with the original queries like `isInteger(1)`, where the input data was known, nondeterminism was used. However, this nondeterminism was hidden away from you. To see how this was hidden, the query below:

```
1 ?- isInteger(2).
2 true.
```

The above query still calls `isInteger`, and it will similarly cause the world to split. In the first world chosen, the fact `isInteger(0)` is used. However, this fact does not work with the given input: we are asking if `isInteger(2)` is true, but instead we are given that `isInteger(1)` is true. These incompatible facts trigger *failure*, the opposite of success. However, this does not completely stop execution, because we see that we have other choices to explore. As such, we then consider the next choice: the fact `isInteger(1)`. This similarly leads to failure, but again there are still choices to explore. Specifically, we still need to explore the fact `isInteger(2)`, which is the next (and last) choice. Upon choosing this fact, the original query of `isInteger(2)` succeeds: this is compatible with the fact `isInteger(2)`.

As described, failure internally occurs twice in execution of this seemingly simple query above, but the query still overall succeeded. The query succeeds as long as **any single** choice works; it is ok (and common) if some choices lead to failure. In this way, failure should not be treated like an error condition; with logic programming languages, failure is a relatively normal part of computation. Failure exists because we don't generally know ahead of time which choice will work, so we may need to explore choices which don't work out.

Only when **all** choices lead to failure does a query fail as a whole. To illustrate this, consider the query below:

```
?- isInteger(3).
false.
```

Internally, this query follows the exact same pattern as previously described. However, since there is no `isInteger(3)` fact, this will eventually fail. In the process, the facts `isInteger(0)`, `isInteger(1)`, and `isInteger(2)` are all attempted, but none of them leads to success.

> ***Sidenote:*** While this whole discussion is true in general, most engines (SWI-PL included) will implement optimizations (such as *clause indexing*) which try to avoid exploring choices which will lead to failure. For example, consider the following query:
>
> ```
> ?- isInteger(0).
> true.
> ```
>
> According to this discussion, the above query should hang, waiting for a semicolon in order to explore additional solutions. However, this will not happen. This is because the engine knows that the other choices for the `isInteger` fact will not work in this case, since there is only one `isInteger(0)` fact. As such, it won't even give you the option to try to explore the other `isInteger` facts: the engine knows they won't work, so it has cut those choices out entirely. Because this is ultimately just an optimization, we do not discuss it in detail.

5 Arithmetic

Arithmetic can be done in Prolog using the built-in `is` keyword. Several basic examples follow:

```
?- W is 4 + 6.
W = 10.
?- X is 3 * 3.
X = 9.
?- Y is 4 / 2.
Y = 2.
?- Z is 10 - 5.
X = 5.
```

Expressions can be nested, as one might expect:

```
?- W is 2 * (1 + 1).
W = 4.
```

Variables can be used in expressions, as long as they have known values. For example, the following is ok:

```
?- X is 2 + 2, Y is X + X.
X = 4, Y = 8.
```

...however, the following closely related query is not ok:

```
?- Y is X + X, X is 2 + 2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

The above query doesn't work because conjunction (`,`) works **strictly** from left-to-right. With this in mind, `Y is X + X` is executed first. This is problematic, as `X` does not have a value until `X is 2 + 2` is executed. This explains the error message, which states that an argument to `is` does not have a known value (specifically `X` in the above example).

Related to arithmetic operations are arithmetic comparisons. Consider the following examples:

```
?- 1 < 2.
true.
?- 1 =< 1.
true.
?- 2 > 1.
true.
?- 2 < 1.
false.
```

```
?- X is 1 + 1, X < 4.
true.
```

As with the arithmetic operations, these arithmetic comparisons only work over known values. For example, if we switch the order of the last query above, an error will result. This is shown below:

```
?- X < 4, X is 1 + 1.
ERROR: </2: Arguments are not sufficiently instantiated
```

In the above example, since conjunction (`,`) works strictly left-to-right, the `X < 4` portion is executed before the `X is 1 + 1` part. Since the variable `X` has no known value at `X < 4`, we get the error message above.

6 Rules

As of yet, we haven't really gotten into how to do much real "work" as far as computation goes. The first example we will use which does real work is that of the factorial function (`!`) from mathematics. Factorial can be expressed as the piecewise function shown below:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

The Prolog code below implements the factorial function above.

```
factorial(0, 1).
factorial(N, Result) :-
    N > 0,
    MinOne is N - 1,
    factorial(MinOne, RestResult),
    Result is N * RestResult.
```

As shown above, we can execute arbitrary code along with a fact by following the fact with `:-`. (This symbol is used because it is reminiscent of reverse implication ($\Leftarrow$), which logically is how calls work. We won't get into those details in this class.) When `:-` is used, it is referred to as a *rule*. Rules are permitted to be recursive, allowing us to perform arbitrary computation.

A line-by-line explanation of the above code follows:

1. Fact implementing the behavior that the factorial of 0 is 1, directly from the original math
2. Rule implementing the recursive case of factorial.
3. Check that the input n is greater than 0
4. Calculation of $n - 1$ in the original math
5. Calculation of $(n - 1)!$ in the original math
6. Calculation of $n \times (n - 1)!$ in the original math

Example queries to the above code are shown below. Note that the engine had additional choices to explore and appeared to "hang" as it did in Section 4. In this case, extra exploration always leads to failure. As such, instead of pressing semicolon (`;`) for more solutions, period (`.`) was instead pressed to stop the search.

```
?- factorial(0, N).
N = 1.
?- factorial(1, N).
N = 1.
?- factorial(2, N).
N = 2.
?- factorial(3, N).
N = 6.
?- factorial(4, N).
N = 24.
```

As shown with the example queries above, the first parameter to `factorial` is the number to get the factorial of. The second parameter holds the result. Note that this works very differently from what you're used to with functional programming. Instead of there being very well-defined inputs and outputs (as there are with functions), this line is intentionally blurred in Prolog. For example, it would have been easy enough above to change the order of the parameters in `factorial`'s definition.

7 Structures

In addition to integers and atoms, we can also form composite data structures. Composite data structures (or just structures) allow us to hold multiple values at once, much like a tuple. The only real difference from a tuple is that they also have a given name associated with them. Compared to SimpleScala, Prolog structures behave like a combination of a constructor and a tuple. To illustrate this, the code snippet below shows a structure named `foo` which contains the values `1` and `2`:

```
foo(1, 2)
```

Structures can be used to build up larger data structures, and often recursive data structures like lists and trees. To demonstrate, we will implement some list operations below which operate on a custom list definition, where `cons` is the name of a structure representing a non-empty list, and `nil` is an atom representing an empty list. With this in mind, we will defined a `myAppend` routine which appends two lists together, resulting in a third list.

```
1 myAppend(nil, List, List).
2 myAppend(cons(Head, Tail), List, cons(Head, Rest)) :-
3   myAppend(Tail, List, Rest).
```

A line-by-line explanation of the above code follows:

1. If the first list is empty, the result (held in the third parameter) should be the same as the second list. In other words, if we append an empty list onto some other list `List`, then the result should be `List`.
2. If the first list is non-empty, name the components of the list `Head` (for the first element of the list) and `Tail` (for the rest of the list). The result list should start with this same `Head`, followed by the other list `Rest`, which has not yet been defined.
3. Recursively call `myAppend`, using `Tail` (the rest of the elements in the first parameter), `List` (the second parameter), and `Rest` (the recursive result).

An example query using the above `myAppend` definition follows. This query appends the list `1, 2, 3` onto the list `4, 5, 6`, yielding the result list `1, 2, 3, 4, 5, 6`. The query follows:

```
1 ?- myAppend(cons(1, cons(2, cons(3, nil))),
2             cons(4, cons(5, cons(6, nil))),
3             Result).
4 Result = cons(1, cons(2, cons(3, cons(4, cons(5, cons(6, nil)))))).
```

Lists are very commonly used in Prolog. As such, there is built-in notation for lists, which helps improve readability and gets rid of all the extra parentheses above. This notation is translated down into normal structures which behave very similarly to `cons` and `nil` above; in fact, the only differences are in the structure names used. Some notes on the built-in list notation follow:

- `[]`: The empty list
- `[1]`: A list containing one element, namely `1`
- `[1, 2]`: A list containing two elements, namely `1` and `2`. This same pattern can be used for a list of length 3 (e.g., `[1, 2, 3]`), and so on.
- `[Head|Tail]`: A non-empty list that starts with the element `Head`, where the rest of the list is held in `Tail`.
- `[First, Second|Rest]`: A list with a minimum length of 2, where the first element is `First`, the second element is `Second`, and the rest of the elements (as in, all elements after `Second`) are in the list `Rest`.

We can rewrite `myAppend` with this updated list notation. This rewrite (yielding `myAppend2`) is shown below, along with the updated query:

```
1 myAppend2([], List, List).
2 myAppend2([Head|Tail], List, [Head|Rest]) :-
3   myAppend2(Tail, List, Rest).
4
5 ?- myAppend2([1,2,3],
6               [4,5,6],
7               Result).
8 Result = [1, 2, 3, 4, 5, 6].
```

While the code above looks very different from the code before, this behaves in exactly the same way.

7.1 Executing "Backwards"

Before concluding this section, there is an important point to make about how `myAppend2` works. In the previous example, we simply appended the lists `[1, 2, 3]` and `[4, 5, 6]` together, yielding the unsurprising result `[1, 2, 3, 4, 5, 6]`. By this point the idea of appending lists together in this way is old-hat, so this does not really show off anything particularly interesting about Prolog.

What is interesting about `myAppend2` is that we can execute it effectively "in reverse". For example, instead of giving it the known inputs of `[1, 2, 3]` and `[4, 5, 6]` and asking for the output, we can instead give a known output of `[1, 2, 3, 4, 5, 6]` and ask for inputs. Such a query is shown below:

```
?- myAppend2(Input1, Input2, [1, 2, 3, 4, 5, 6]).
```

If we keep hitting semicolon (`;`) and ask for different query results, we end up with 7 in all, listed below:

1. `Input1 = [], Input2 = [1, 2, 3, 4, 5, 6]`
2. `Input1 = [1], Input2 = [2, 3, 4, 5, 6]`
3. `Input1 = [1, 2], Input2 = [3, 4, 5, 6]`
4. `Input1 = [1, 2, 3], Input2 = [4, 5, 6]`
5. `Input1 = [1, 2, 3, 4], Input2 = [5, 6]`
6. `Input1 = [1, 2, 3, 4, 5], Input2 = [6]`
7. `Input1 = [1, 2, 3, 4, 5, 6], Input2 = []`

As shown with these results, such a query effectively asks "which two unknown lists, when appended to each other, yield the list `[1, 2, 3, 4, 5, 6]`?" Each of the above answers reflects some possible combination of the input lists `Input1` and `Input2` to yield this expected result list (e.g., the first result appends the empty list onto `[1, 2, 3, 4, 5, 6]`, the second result appends the list `[1]` onto the list `[2, 3, 4, 5, 6]`, and so on). In this way, depending on the query we issue to `myAppend2`, we can effectively execute "backwards", going from known outputs to unknown inputs.

8 Unification

While Section 3 discussed variable usage at length, and variables have been used in each subsequent section, a full discussion of how variables get values has never been provided. These details are important for understanding how Prolog works overall, particularly in the context of "backwards execution" in Section 7.1. We shed some insight on exactly how variables get values in this section.

In Prolog, variables are assigned values via *unification*. Unification is effectively mathematical equality (=), but in a way that can be implemented within the Prolog engine. In fact, `=` is used to tell the Prolog engine to unify two values, just as we would say that two values should be equal in math. Like mathematical equality, unification operates in either direction. For example, consider the following two queries:

```
1 ?- X = 1.
2 X = 1.
3 ?- 1 = X.
4 X = 1.
```

Both of these queries result in the assignment of `1` to `X`, and they fundamentally work in the exact same way. This is decidedly **unlike** the assignment operator from other languages, wherein the variable must be on the left and the value must be on the right. This is somewhat obnoxious because other languages still use the syntactic `=` for assignment, even though assignment bears little connection to its mathematical meaning. In contrast, Prolog's unification operator works just like mathematical =, and Prolog lacks assignment.

> ***Sidenote:*** Strictly speaking, there *is* a way to perform true assignment in Prolog, but not with the `=` operator. We will not get into these features in this class, which require global variables and mutable state.

Also like mathematical = and unlike assignment, once a variable has a value, the value of the variable can never change. For example, the following query fails:

```
1 ?- X = 1, X = 2.
2 false.
```

At the point of `X = 1` in the above query, Prolog stores that the value of variable `X` is `1`. When `X = 2` is encountered, this ultimately triggers failure (leading the engine to report `false` above), as `1` and `2` are not equal to each other.

The related query below does, however, succeed:

```
1 ?- X = 1, X = 1.
2 X = 1.
```

At the first use of `X = 1`, the value of variable `X` is stored to be `1`. At the second use of `X = 1`, we simply check to see if the stored value of `X` is `1`. In this case, the stored value of `X` is equal to `1`, and so the query succeeds with the information that the value of `X` is `1`.

This process is easy enough to follow when we deal with exactly one variable and one value. However, part of the power of Prolog is that we can reason about many variables and values. For example, consider the following query:

```
1 ?- X = Y, X = 1, Y = 2.
2 false.
```

The above query fails because we first added a constraint that `X` and `Y` should be the same value, and then we tried to give `X` and `Y` separate values (namely `1` and `2`, respectively). If we had never issued the `X = Y` part in the query above, this query would have succeeded. Such a modified query can be seen below, which succeeds:

```
1 ?- X = 1, Y = 2.
2 X = 1, Y = 2.
```

What the above two queries show is that somehow we need to record if two variables hold the same value, **even if** we don't know exactly what that value is yet. This was the case for the `X = Y` portion above, which recorded that variables `X` and `Y` should hold the same value, even though values had not been given to `X` and `Y`.

This sort of recording can be most easily understood in terms of *equivalence classes*. Roughly, an equivalence class is a set of elements which are all declared to be equal. Initially, all values and variables are in their own separate equivalence classes. For example, let's say that we have variables `X` and `Y`, along with the value `1`. Initially, there are three equivalence classes: one containing `X`, a second containing `Y`, and a third containing `1`. If we issue a unification `X = Y`, this effectively states to *merge* the equivalence classes for `X` and `Y`. This merge leaves us with two equivalence classes: one containing both `X` and `Y` (hereinafter named $X \cup Y$), and a second containing the value `1`. If we then issue the unification `X = 1`, we will then take the equivalence class containing `X` (namely $X \cup Y$) and merge it with the equivalence class which just contains `1`. In this example, the end result is a single equivalence class holding variables `X` and `Y`, along with the value `1`. The semantic meaning of this is that variables `X` and `Y` share the same value, and that value is `1`. For the rest of the discussion, we will call this equivalence class $X \cup Y \cup 1$.

Care must be taken when merging equivalence classes. For example, if we take the equivalence class $X \cup Y \cup 1$ and attempt to merge it with an equivalence class containing `2`, this should fail: `1` does not equal `2`, and so we can never merge equivalence classes containing different values. In Prolog, such faulty attempts to merge (i.e., attempts to unify values which do not unify) lead to failure. This failure will either cause the entire query to fail, or at least go back to whatever the last choice was and (hopefully) try to unify with a different value.

In the last assignment, you will implement this described unification operation. At that point, we will discuss all the nitty gritty details involved in a basic implementation of this operation. For now, this explanation works to understand the basics of unification, though it's probably not enough information to go on to implement the operation.