A Brief Introduction to Prolog

1 Overview

We will be using Prolog for the next part of this course. Prolog, short for "**Pro**gramming in **log**ic", is a *logical language*, in contrast to *object-oriented languages* like JavaScript and *functional languages* like Swift. True to the name, 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 course 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, for my own research, 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, often incomplete 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, I 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 from the command-line, which will bring you to the REPL (Read-Eval-Print-Loop: a place where you can type in code and have it be evaluated on the fly). 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 (kind of like `main` in most other languages).

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:

```
1 ?- isInteger(3).
2 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:

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

Expressions can be nested, as one might expect:

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

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

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

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

```
1 ?- Y is X + X, X is 2 + 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

So far, the queries we have written have tended to accomplish more useful work than the actual code. We can put more code in our code by using *rules*. For example, let's say we want to encode the idea that a given input is between 7 and 10, exclusive. This can be expressed using the following Prolog rule:

```
between7And10Exclusive(N) :-
  N > 7,
  N < 10.
```

As shown above, we can execute arbitrary code along with a fact by following the fact with `:-`. When `:-` is used, it is referred to as a *rule*. This symbol is used because it is reminiscent of reverse implication ($\Leftarrow$), which logically is how calls work. With this in mind, we can rewrite the above code in the following logical way:

$$\texttt{between7And10Exclusive}(n) \Leftarrow (n > 7 \wedge n < 10)$$

Alternatively, if you prefer the non-reversed implication:

$$(n > 7 \wedge n < 10) \Rightarrow \texttt{between7And10Exclusive}(n)$$

That is, if a number n is greater than 7 and less than 10, this logically implies that the predicate `between7And10Exclusive`(n) is true.

With the above rule in hand, we can issue queries to the rule. Some example queries are shown below.

```
?- between7And10Exclusive(6).
false.
?- between7And10Exclusive(7).
false.
?- between7And10Exclusive(8).
true.
?- between7And10Exclusive(9).
true.
?- between7And10Exclusive(10).
false.
```

In practice, rules in Prolog are used much like functions or methods in other languages. Much like functionds and methods, rules allow us to execute some programmer-defined computation whenever we want, and this computation can be parameterized by different values. In terms of thinking in Prolog, whenever you're in a situation where you want to write a function or a method, you probably want to write a rule.