Notes for Lecture 17 (Fall 2022 week 8, part 1): Introduction to Prolog

Jana Dunfield

October 30, 2022

The code for this lecture is in `lec17.pl`.

These notes, with `lec17.pl`, may take two lectures to cover, not one.

Because we have Quiz 2 on Thursday (2022-09-03), Wednesday's lecture will be a review of *Haskell* for the quiz. We'll return to Prolog in Week 9.

1 SWI-Prolog

1.1 Installation

Download SWI-Prolog from the website

https://www.swi-prolog.org/download/stable

Installing SWI-Prolog is usually smoother than installing GHC, but please ask on Piazza if you run into any trouble.

1.2 Beware other Prologs

Haskell is effectively standardized, because there is one major implementation of Haskell: GHC. Prolog is not standardized. If you install other things called Prolog, they will not always work the same way as SWI-Prolog.

You may wonder why I'm telling you this, because if I tell you to install SWI-Prolog you'll probably install it and not something else you find. That's not what I'm worried about. The real problem is that materials you find online, *and even parts of the optional textbook for Prolog* (Bratko, *Prolog Programming for Artificial Intelligence*), may be about other versions of Prolog. That can be frustrating, so if you're looking for other material to read, keep in mind that it may not apply to SWI-Prolog.

(I learned Prolog in the 1990s and barely touched it until I taught this course. Discovering that some of the textbook's examples just don't work in SWI-Prolog was an unpleasant surprise, especially since the textbook used to be required in this course. I can't make it pleasant, but at least it won't be a surprise.)

1.3 Loading Prolog files

The way I load a Prolog file in SWI-Prolog is by typing, at the "`?-`" prompt, something like

```
consult('/Users/jana/360/lec17.pl').
```

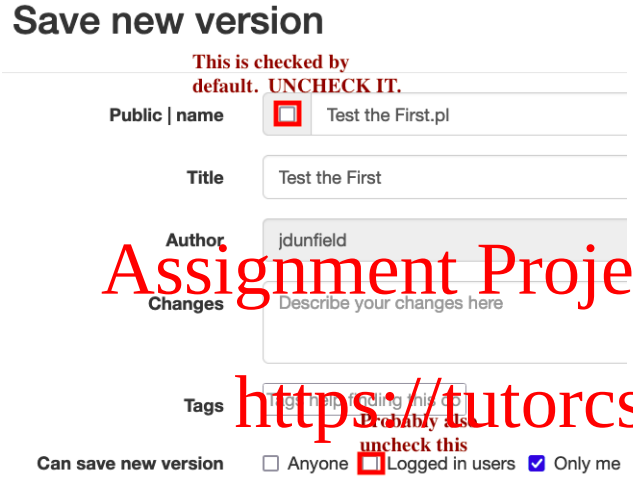
The specific string depends on your username and where you store course material. It also depends on the pathname syntax for your OS, so you might need to use different syntax on Windows.

Another way (at least for *some* versions of macOS) is to select "Consult" from the File menu, and choose the file you want to load.

1.4 Alternate method

There is a public website called **swish.swi-prolog.org**. I can't officially recommend this: it's slow, somewhat unreliable, and encourages you to make your code visible to the entire world.

If you do use it anyway, I suggest (1) making an account on the site (based on a StackOverflow account) and (2) unchecking some boxes when you save programs:



2 Prolog: why?

Prolog is a *logic programming* language. It takes two main ideas and pushes them pretty far:

- backwards proof search (CISC 204's "backwards reasoning");
- unification (basically, non-mathematical equation solving).

Decades ago, when "artificial intelligence" was a buzzword (just like now), AI was more focused on "deductive reasoning" than on "machine learning". Put very roughly, the foundation was CISC 204, not statistics.

Each language paradigm (functional, logic, object-oriented, etc.) makes some things easier and some things harder. Assignment 3, for example, would probably be more annoying to do in Java or Python than in Haskell. (At least, assuming roughly equal levels of familiarity with Java/Python and Haskell.) But much of Assignment 3 fits very nicely into Prolog. Most of the complicated structure of the Tiny Theorem Prover goes away in Prolog.

On the other hand, some of the wilder AI dreams of Prolog don't pan out in reality. The temptation Prolog offers is that we can directly translate logical rules into code, run them, and get what we want. In practice, while we *can* directly translate logical rules into Prolog code and run them, the result is not always what we want. Infinite recursion is a frequent problem in Prolog code, as is code that takes a very long time to run. This is partly because Prolog is ambitious: in a sense,

it proposes to do all your CISC 204 tests automatically. But answering some of those questions on the 204 tests required thought, and we don't know how to automate thought. Prolog goes looking for solutions without thinking, and it may try paths that don't lead anywhere.

2.1 Prolog: probably not what you're used to

People often find Haskell to be unfamiliar overall, but some parts of Haskell are pretty similar to other languages you've learned. For example, arithmetic in Haskell is quite similar to arithmetic in Python or Java. The names given for the same things (or similar things) may vary, but a Haskell "function" is fairly close to a Java "method": each takes inputs (called "arguments") and returns a result. (There are differences. For example, Java methods don't have to return a result, and Haskell functions always do; Java methods have an object `this` which is an implicit argument, and Haskell functions don't.)

Prolog is more different. A Prolog program is not executed or evaluated. A Prolog program expresses logical information—a "knowledge base" of *facts* and *rules*. A fact is something that we assume is true, and a rule is a way to derive new facts from other facts.

By themselves, facts and rules don't do anything. The utility of Prolog comes from the "query engine" built into the Prolog compiler, which lets us ask questions (queries) about the facts and rules. This query engine is an algorithmic (in the sense of "precise instructions to do something", not "statistical guessing") version of the *backwards reasoning* taught in CISC 204. When we ask Prolog whether something is true, say, "is MATH 101 a prerequisite to MATH 102", it looks through the facts and rules in the program. If it sees a matching fact, it says "yes, this is true". If it doesn't, it looks for a rule whose conclusion matches "is MATH 101 a prerequisite to MATH 102", and then tries to derive the premises.

(Compare to CISC 204, where you learned that one way to prove a conjunction $\varphi_1 \wedge \varphi_2$ is to use the "and-introduction" rule. Once you've decided to use that rule, your goal is to prove the two premises, φ_1 and φ_2.)

We will look at this process in much more detail later, so don't worry if the above seems murky.

3 Clauses: facts and rules

Before we do that, we need to become somewhat comfortable with the way that facts and rules are expressed in Prolog.

A Prolog program consists of *clauses*. A clause is either a fact—something that we're saying is always true, it doesn't depend on anything else being true—or a rule.

3.1 Facts

First, let's look at some facts we could write (see lec17.pl):

```
has_elevator(goodwin).
has_elevator(beamish_munroe).
```

The words `has_elevator` and `prereq` are used as *predicates*. This is the same idea as in CISC 204, but we will be less terse. In 204 we might write "$E(g)$" to mean that the predicate E (for **E**levator), holds of the object g (for **G**oodwin). In Prolog we generally write out the entire word, or at least a reasonable abbreviation, like `prereq` for "prerequisite".

The first two facts above say that Goodwin has an elevator, and Beamish-Munroe has an elevator. The words `goodwin` and `beamish_munroe` correspond to Haskell data constructors. Unlike Haskell, in Prolog we don't need to declare them as part of a specific 'data' type; we actually don't need to declare them at all. Prolog's capitalization rules, annoyingly, are exactly backwards from Haskell's: in Haskell, data constructors must begin with a capital letter, and in Prolog, data constructors must begin with a *lowercase* letter.

As in 204, predicates can talk about more than one thing. Saying that "MATH 101 is a prerequisite" doesn't mean anything by itself; we need to know which course it's a prerequisite *for*. We can model this with a binary predicate `prereq`:

```
prereq(math101, math102).
prereq(math101, chem101).
prereq(math102, math210).
prereq(math102, phys102).
prereq(phys102, phys201).
prereq(math210, phys201).
```

In 204 we might have compressed the first line into something like "P(m101, m102)", but again, the idea is similar; `prereq(X, Y)` relates two "objects" (courses) by saying that you have to take course `X` before you can take course `Y`.

By the way, I have no idea if these are actual course prerequisites at Queen's. As usual with computers, the answers Prolog gives are only as good as the information it has; if we tell it that CISC 360 was needed *before* CISC 204, it would assume that to be true.

Exercise 1. Add the (correct) fact that says CISC 204 is a prerequisite of CISC 360 to `lec17.pl`.

3.2 A general warning about Prolog

Haskell has a type checker that complains when you do something that doesn't make sense. Sometimes this feels like being yelled at, but the things Haskell complains about actually don't make sense; if you have enough experience to understand its error messages, they can really help you write a correct program.

Prolog, generally, doesn't complain. I really don't like this; I want to be told when I'm doing something that's wrong, or even probably wrong.

For example, if we were trying to say that the Stauffer and Mitchell buildings have elevators, and wrote

```
has_elevator(stauffer, mitchell).
```

instead of

```
has_elevator(stauffer).
has_elevator(mitchell).
```

Prolog would not complain. It's pretty dubious to use the same name `has_elevator` for a unary predicate (one argument) and a binary predicate (two arguments), but Prolog tends to assume that you meant to say whatever you said, so it allows this.

3.3 Another warning about Prolog

You have all used more than one programming language, so you know that different languages have different notations for things. We haven't seen enough of Prolog yet to see this, but Prolog's conventions often differ from Haskell's—sometimes they're *exactly opposite* to Haskell's. So don't assume that things we've learned for Haskell, like "data constructors have to start with an uppercase letter", hold for Prolog. Prolog enforces the opposite: the thing in Prolog that's like a data constructor must begin with a *lowercase* letter.

Prolog also has something like Haskell's pattern matching, and again, enforces the opposite convention from Haskell. In Haskell, pattern variables must begin with a lowercase letter. In Prolog, pattern variables ("unification variables") must begin with an uppercase letter.

3.4 Simple queries

Before talking about rules, let's try to confirm that Prolog actually loaded the facts in the file. Start SWI-Prolog and type the following (the `?-` will already be there). You will need to change the pathname to the location of `lec17.pl` on your computer. You can also try to select "Consult" from the "File" menu, and navigate to the file's location on your computer.

```
?- consult('/Users/jcunfield/360/lec17.pl').
```


This may produce some warnings about "singleton variables". These are warnings, not errors; we'll talk about what they mean later. As long as the last line displayed is "**true.**", the file was loaded successfully.

Let's ask if Goodwin has an elevator:


```
?- has_elevator(goodwin).
true.
```

Prolog answers "true" because it found that exact fact in `lec17.pl`.

Now let's ask if Walter Light has an elevator:

```
?- has_elevator(walter_light).
false.
```

A more powerful query is to ask "does there exist a building that has an elevator?" We do this by writing an uppercase letter instead of the building name:

```
?- has_elevator(X).
X = goodwin
```

Here, Prolog pauses to wait for instructions: are we satisfied with this answer, or do we want to find other buildings that have elevators?

If we type a period, Prolog stops looking and waits for a new query:

```
?- has_elevator(X).
X = goodwin .

?-
```

If we type a semicolon, Prolog keeps looking. There are only two facts about elevators in `lec17.pl`, so (in this particular situation) Prolog knows there are no more buildings with elevators, so it stops.

```
?- has_elevator(X).
X = goodwin                (I typed  ;  here)
X = beamish_munroe.

?-
```

Here is a slightly more complicated query: "Is there a course `X` which is a prerequisite for `phys201`?" We can translate this question into a 204-like formula:

$$\exists \texttt{X}\,(\texttt{prereq(X, phys201)})$$

Prolog doesn't make you write ∃X—it assumes that if you write an uppercase letter, you're asking whether such a thing exists. So we enter `prereq(X, phys201)`:

```
?- prereq(X, phys201).
X = phys102 .

?-
```

I decided I only wanted one answer, so I typed a period.

Exercise 2.

1. Find the other prerequisite of `phys201`.
2. Find the "follow-on" courses of `math102`: that is, for which course(s) `Y` is `math102` a prerequisite?

3.5 Rules

With only facts, Prolog is mildly useful (since we can write queries involving `X` and `Y`) but not really that interesting. Rules allow us to express more general reasoning.

A rule looks like

```
conclusion :- premises.
```

Think of the `:-` symbol as ⟵: "if the premises are true, then the conclusion is true".

I often write rules on several lines, like this:

```
conclusion :-
  premise1,
  premise2.
```

We are going to define a predicate `required` that tells us about indirect prerequisites. According to the facts in `lec17.pl`, you have to take `math101` before `math102`, and you have to take `math102` before `phys102`. So `math101` is an *indirect* prerequisite of `phys102`, because `math101` is a (direct) prerequisite of `math102` and `math102` is a (direct) prerequisite of `phys102`.

We are already using `prereq` to mean "direct prerequisite". We'll use `required` to mean "indirect prerequisite".

There are two ways that a course `A` can be an indirect prerequisite of `C`:

- `A` can be a direct prerequisite of `C`.
- `A` can be a direct prerequisite of some other course `B`, where `B` is an indirect prerequisite of `C`.

If we got to write our own rules in 204, we could express these as:

$$\frac{\texttt{prereq(A, C)}}{\texttt{required(A, C)}} \qquad \frac{\texttt{prereq(A, B)} \quad \texttt{required(B, C)}}{\texttt{required(A, C)}}$$

Just as in 204, we can replace "meta-variables" like φ and ψ with formulas, Prolog will replace `A`, `B`, and `C` with things like `math101`.

In Prolog, we write the rules like this:


```
required(A, C) :-
  prereq(A, C).

required(A, C) :-
  prereq(A, B),
  required(B, C).
```


Let's translate these back into (mathematical) English:

```
required(A, C) :-
  prereq(A, C).
```

"`A` is required for `C` if `A` is a prerequisite for `C`".

If we prefer to think of the implication the other way, we could say

"If `A` is a prerequisite for `C`, then `A` is required for `C`."

However we think about the implication, Prolog has "hidden quantifiers": the rule

```
required(A, C) :-
  prereq(A, C).
```

corresponds to the logical formula

$$\forall \texttt{A}\, \forall \texttt{C}\, (\texttt{prereq(A, C)} \rightarrow \texttt{required(A, C)})$$

Translating our second rule to a logical formula would give:

$$\forall \texttt{A}, \texttt{B}, \texttt{C}\, (\texttt{prereq(A, B)} \wedge (\texttt{required(B, C)} \rightarrow \texttt{required(A, C)})$$

(When I write $\forall X, Y, Z$ I mean $\forall X\ \forall Y\ \forall Z$.)

Translating this back into (mathematical) English, we get:

For all A, B, C,
if A is a direct prerequisite of B
and B is an indirect prerequisite of C, then A is an indirect prerequisite of C.

Exercise 3. Write an appropriate Prolog rule for a predicate `postreq(X, Y)` such that `postreq(X, Y)` is true if and only if `Y` is a direct prerequisite of `X`.

Test your rule with some queries.

Finally, translate your rule into (1) a logical formula and (2) into mathematical English.

(If you really want, also translate it into a 204-style rule with a horizontal line, but I don't especially recommend this because it's helpful to see the ∀s written out. 204-style rules implicitly say "for all formulas φ" but we don't write the "for all".)

4 Modelling data in Prolog

You should probably read 3.2 again. I wish you didn't have to.

Consider a Haskell 'data' declaration

```
data Tree = Empty
          | Node Tree Integer Tree
```


This models trees with empty leaves and (branch) nodes containing Integer keys.

In Prolog, we can't really declare this as a type. The best we can do (for now) is "declare" the type in a comment:


```
% A tree is one of:
%   empty.
%   node(A, K, B).   where A is a tree.
%                    K is an integer,
%                    B is a tree.
```


This is only a comment, so Prolog will not enforce it—it will let us write things like `node(empty, empty, empty)`, even though `empty` is not an integer.

What I really detest, though, is that Prolog will let us write `node` as if it were a predicate. It's not a predicate—it's a data constructor. But Prolog will let us accidentally use it as predicate. For now, I'll use it as a data constructor and hope you don't get too confused. (Rant over.)

To write the equivalent of the Haskell expression

```
Node Empty 3 Empty
```

we can write `node(empty, 3, empty).`

```
%         node
%        / | \
%       /  |  \
%   empty  3   empty
```

Let's define a predicate `child` that tells us if one tree is a child (either left or right) of another tree.

This requires only two facts:

```
child(L, node(L, K, R)).

child(R, node(L, K, R)).
```

As with rules, if we write uppercase letters like `L`, they will be universally quantified: the first fact means ∀`L,K,R` (`child(L, node(L, K, R))`).

5 See lec17.pl

The remaining parts of this lecture are in comments in `lec17.pl`, because I didn't have time to typeset them nicely.