Introduction
Prolog is a *logical* programming language. This means that programs written in Prolog use the idea of formal logic. Specifically, a Prolog program is a collection of *facts* and *rules,* which build a kind of database which we can then query. Prolog is highly *declarative* rather than *imperative*, which is often one of the challenges for programmers coming from more imperative languages. Prolog can be useful in solving some difficult problems precisely because the programmer does not have to specify an algorithm for solving it. Instead, we use rules for *describing* what is true about the solution, and we let the Prolog system do the rest by searching for a solution that fits the description.

Using Prolog
We will use the interactive SWI-Prolog system, which is available on lectura. You can also download it yourself from https://www.swi-prolog.org. It is also possible to find online Prolog editors.

Basics
We aren't going to worry much about *types* in Prolog, though there are technically different types of data, including integers, floating point values, and lists. The thing is, we don't have to specify types or worry much about type inference because Prolog is all about *pattern matching*, which we call *unification.* We will discuss that process in more detail later. The good news is it works similarly to the pattern matching we used in SML.

Instead of dealing with various *types*, we will think about *terms*. Everything in Prolog is a *term,* which can be

- a *constant,*
- a *variable,*
- or a *compound term.*

A *constant* can be

- an integer such as `14` or `-8`,
- a real number such as `3.141592` or `-1.23e5`,
- or an *atom*.

An *atom* starts with a lower-case letter and has 0 or more letters, digits, or underscores after it. Keep in mind that *atoms* are constants, not variables.

Here are some examples of *atoms*:

- `parent`
- `fred`
- `are_friends`
- `child`

Here are some *atoms* that may be less obvious because they appear to be operators. These can serve as operators in a way (which we will get to later), but it's important to understand that they are just *constants* that will be used in the *unification* process: `+, *, =, /`
A *variable* starts with an uppercase letter or an underscore, but the ones starting with underscores are special and get special treatment. In general, when you write your Prolog programs, it's a good idea to start your variables with an uppercase letter.

Here are some examples of variables:

- `X`
- `Child`
- `_123`
- `Parent`

A *compound term* is an *atom* followed by *a list of terms*.
These often look like function calls, *but they are not.*

Here are some examples:

- `x(y,z)`
- `parent(adam, Child)`

Unification–a brief overview
Before we talk about what unification *is* and how Prolog works, we should clarify a few things about how Prolog *does not work:*

- We are not dealing with functions. The compound terms may look like function calls with arguments passed in, but that is not what they are. They are not returning a value. All that is happening here is binding of variables with pattern matching.
- We are also not dealing with variable *assignment*. Variables are bound to values through unification.

A first Prolog program.
A Prolog program is a collection of *facts* and *rules* that create a kind of *database* that can be used to find answers to queries. The first program we will look at has to do with a family tree, and as a simple example, I am using the Addams family since it is small and easy to find online.

We start with a list of facts about who is a parent of whom. These facts are stored in a file called `addams.pl`.

```
parent(grandmama,pancho).
parent(grandmama,gomez).
parent(hester,morticia).
parent(hester,ophelia).
parent(unknown,hester).
parent(unknown,fester).
parent(gomez,pugsley).
```

```
parent(gomez,wednesday).
parent(morticia,pugsley).
parent(morticia,wednesday).
```

What we've done here is created a family tree using only one relationship–parents and children. The meaning of these facts is as follows: `parent(X,Y)` means **"`X is a parent of Y`".**
But it is important to note that to Prolog, this doesn't really *mean* anything in particular. In other words, the atom "parent" has no particular meaning in Prolog–any more than a variable named "size" means anything particular in Java. (But again, I should be careful here because "parent" is not a variable here, but a constant.)

To start up the SWI-Prolog system, type the following into your terminal and press Enter: `swipl`
Now, we can type in Prolog queries.
The first one we will do uses a special term to bring in files:

```
?- consult(addams).
true.
```

Notice that I did not include the filename extension. Also, note that the "output" from this is "true". This tells us that it worked and now all the facts from that file are in the database.

Now, we can make some queries.

```
?- parent(grandmama,pancho).
true.
```

What is happening here is all *unification* (or pattern-matching). I type in my query and Prolog searches the database to find a match. Since it does find a match for this exact compound term, the result is "true".

See what happens if I try something that does not have a match:

```
?- parent(grandpapa,pancho).
false.
```

But of course these are not particularly interesting or useful queries since they only contain constants, which will only match with exact constants.
So let's try using a variable instead:

```
?- parent(X,pancho).
X = grandmama.
```

Here, we do a more open-ended query where we are looking for something to bind with `X`, which will make `parent(X,pancho)` a true statement. Prolog searches and finds that `parent(grandmama, pancho)` is in the database, so it binds `X` to `grandmama`.

Here's another example:

```
?- parent(X,wednesday).
X = gomez
```

Here, though, the query holds, waiting for another input. This is because Prolog will keep searching for more answers if you let it. If you press Enter here, it will end, leaving you with the one binding for X. But if you press the semi-colon (;) key, it will keep searching, and in this case, it will find another possible binding for X. (Because Wednesday has two parents.)

```
?- parent(X,wednesday).
X = gomez ;
X = morticia.
```

We can also make queries where we know the parent, but not the child:

```
?- parent(pancho, X).
false.
?- parent(gomez, X).
X = pugsley ;
X = wednesday.
```

Or even make a query to give us all parent-child pairs:

```
?- parent(P,C).
P = grandmama,
C = pancho ;
P = grandmama,
C = gomez ;
P = hester,
C = morticia ;
P = hester,
C = ophelia ;
P = unknown,
C = hester ;
P = unknown,
C = fester ;
P = gomez,
C = pugsley ;
P = gomez,
C = wednesday ;
P = morticia,
C = pugsley ;
P = morticia,
C = wednesday.
```

Writing Rules.
Just querying parent facts, is not super useful or interesting. But we can also write rules to infer new relationships from the parent facts.
For example, we can write a fact that uses parent facts to determine who is a grandparent of whom.

```
% grandparent(X,Y): X is a grandparent of Y
grandparent(X,Y) :- parent(X,P), parent(P,Y).
```

Explanation. First, let me note that `%` indicates the beginning of a comment.
Next, let's talk about the grandparent rule.

Recall that Prolog is *declarative*. We write rules for new predicates by describing what has to be true in order for that predicate to be true. We can indicate that multiple things must be true by writing multiple *goals*, which are separated with commas. You can think of these commas as logical ANDs because in order for the defined predicate to be true, *all the goals must be true.*

So we describe what it means for X to be a grandparent of Y by using the parent predicate. We indicate that X would be a parent of some person P, and that person would be a parent of Y.

Here is another example, which indicates the X is a sibling of Y if they share a parent.

```
sibling(X,Y) :- parent(P,X), parent(P,Y), not(X=Y).
```

You can also think of rules as "proofs". To "prove" that X and Y are siblings, we prove that X and Y have a parent in common. We also check that they are not the same person because without that, a person would be indicated as their own sibling (since technically they share a parent with themselves).

The sibling predicate illustrates an important point. Sometimes the order of the goals doesn't really matter because they all have to be true either way. But sometimes it does matter–either for correctness or for efficiency or both.

In this case, the order matters for the sake of getting correct results, and it has to do with the order that variables are bound. We can see this by tracing a query.
SWIPL allows you to do this with `trace.` In blue below is my own added explanation.

```
?- trace.
true.
[trace]  ?- sibling(pansy,X).
   Call: (10) sibling(pansy, _28130) ? creep
      The _28130 is an intermediary variable used in the search process.
   Call: (11) parent(_30234, pansy) ? creep
      The first goal is to find a parent of pansy.
```

```
      The _30234 is an intermediary variable used in the search.
   Exit: (11) parent(balbo, pansy) ? creep
      Searching the database, Prolog finds a match and binds the variable to
balbo, which is the first parent listed for pansy.
   Call: (11) parent(balbo, _28130) ? creep
      The next goal is to find a child of balbo.
   Exit: (11) parent(balbo, mungo) ? creep
      Prolog finds mungo.
^  Call: (11) not(pansy=mungo) ? creep
      Now that the intermediary variables are bound, it checks that pansy and
mungo are not the same.
^  Exit: (11) not(user:(pansy=mungo)) ? creep
   Exit: (10) sibling(pansy, mungo) ? creep
      Since they are not, all the goals pass and the predicate is true with
these variable bindings.
X = mungo .
      X is bound to mungo because that binding makes the predicate true.
```

If we change the sibling fact so that the last goal is first, we will see very different results.

```
sibling(X,Y) :- not(X=Y), parent(P,X), parent(P,Y).
```

In this case, the query will fail. This is because `X` is bound to `pansy` and `Y` is not yet bound when `not(X=Y)` is checked and since `Y` can be bound to `pansy` and `pansy = pansy` is true, the goal will fail because `not(pansy=pansy)` is false.

This is a case where a predicate does not work so well when the variables are not yet bound.

Note. You can turn off the trace with the following:

```
[trace]  ?- notrace.
            true.
[debug]  ?- nodebug.
            true.
```

Multiple Rules & Recursion.

As noted before the "," indicates the logical AND indicating that there are multiple goals that must be true for the predicate to be true.

We can indicate the logical OR by writing multiple rules for the same predicate. Then, if Prolog fails to find a solution with the first rule, it will try the next one.

We can also define rules recursively. Using the family database, we can write a rule to infer when X is an ancestor of Y.

Basically, X is an ancestor of Y if:

- X is a parent of Y (the base case) or
- X is an ancestor of Y's parent (recursive case)

In Prolog:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(P,Y), ancestor(X,P).
```

Lists.

Prolog uses singly linked lists just like SML. These lists also have no tail pointer, just like SML. It is also easiest to deal with these lists using pattern matching. Here are some example list-related queries with some explanation.

```
?- X = [1,2,3,4].
X = [1, 2, 3, 4].
X is bound to the list [1,2,3,4]
?- [X|T] = [1,2,3,4].
X = 1,
T = [2, 3, 4].
The | symbol divides elements in the beginning of a list from the rest of the
list. Here X is bound to the head of the list (1) and T is bound to the tail
([2,3,4]). It is similar to the :: operator in SML.
?- [X,Y|T] = [1,2,3,4].
X = 1,
Y = 2,
T = [3, 4].
Here, the pattern shows two elements before the |, so it will match with any
list that has at least two elements. In this case, X is bound to 1, Y is
bound to 2, and T is bound to the rest of the list ([3,4]).
?- [X,Y,Z,A|T] = [1,2,3,4].
X = 1,
Y = 2,
Z = 3,
A = 4,
T = [].
In this case, all the elements in the list are bound to specific variables,
and the "tail" is empty.
```

Special terms and "operators".

Prolog is all about unification (aka pattern matching), so there is no variable assignment and every term is just a term that can be used with pattern matching. Even operators are terms and can be treated as constants in pattern matching.

However, they can also be evaluated if they are used with the right term. Here are some examples with explanations.

```
?- X = 3*4.
X = 3*4.
```
The "=" term does no evaluation. It only does pattern matching, so here X is simply bound to the term 3*4.
```
?- X is 3*4.
X = 12.
```
The "is" term will evaluate the right hand side before doing pattern matching, so here X is bound to 12.
```
?- 3*4 =:= 10+2.
true.
```
The "=:=" term will evaluate both sides first, then do pattern matching, so this evaluates to 12 and 12 and then does pattern matching. Since 12 = 12, this is true.
```
?- 3*4 = 10+2.
false.
```
This fails because these terms (without evaluation) do not match.
```
?- 3*4 is 10+2.
false.
```
This also fails because only the right side will be evaluated before pattern matching and 3*4 does not match 12.
```
?- 12 is 10+2.
true.
```
This works because 10+2 will be evaluated to 12 and then pattern matching will match 12 to 12.

The following terms can be used in this way as arithmetic operators: +, -, *, /
The following terms can be used to compare integers: <, >, =<, >=

List Predicates.
The following are some list predicates that are built into Prolog. You should try these out in various queries to get comfortable with them. Note that some are more flexible than others. For example, *append* is quite flexible and can handle many different queries, but *sort* is less flexible because it will not *unsort* a list.

Please remember that these are predicates that are either *true* or *false* depending on the parameters. They are not functions!

- `append(X,Y,Z)`: Z is Y appended to X
- `select(X,Y,Z)`: Z is Y with one instance of X removed
- `nth0(X,Y,Z)`: Z is the Xth element in Y where indexing starts at 0
- `nth1(X,Y,Z)`: Z is the Xth element in Y where indexing starts at 1
- `length(X,Y)`: Y is the length of list X
- `reverse(X,Y)`: Y is the reverse of X

- `sort(X,Y)`: Y is X, sorted
- `member(X,Y)`: X is a member of Y