In this lecture, we will explore *logic programming* in the language Prolog. We will use a freely available implementation called SWI-Prolog, which is installed on the department’s Linux workstations as `/usr/bin/prolog`.

When we start Prolog, we get a prompt:

```
?-
```

This prompt indicates that the environment expects a *question* from us.\(^1\) For example, we might ask:

```
?- 3 = 3.
```

Yes

Or, if we ask:

```
?- 3 = 2.
```

No

Unfortunately, Prolog doesn’t have much built in for us to work with, so we’ll need to write our own programs before we can ask more interesting questions. We’ll start with a simple example—family trees. In Prolog, we can represent each edge in the tree with a single statement. For instance:

```prolog
parents(greg, bill, joyce).
parents(bill, harold, natalie).
parents(joyce, henderson, doris).
...
```

(Note that we put a period at the end of each statement, just like after each question above.) Each statement is called a *fact*, and together they comprise a simple Prolog program. We can now load this program into the Prolog environment.

```
?- consult('tree.prolog').
% tree.prolog compiled ...
```

Yes

Once we’ve loaded it, we can ask questions about it:

```
?- parents(greg, joyce, bill).
```

No

```
?- parents(greg, bill, joyce).
```

Yes

\(^1\)This is different from DrScheme’s Interactions Window, or Haskell’s REPL, where the user enters an *expression* for evaluation.
These simple yes/no questions are nice, but we'd really like to be able to ask more sophisticated questions like “who are Bill’s parents?” Prolog gives us a way of asking such questions, which involves the use of variables. In Prolog, any identifier that begins with an upper-case letter or underscore is a variable, so we can simply say:

?- parents(bill, Father, Mother).

Father = harold
Mother = natalie

Yes

The question essentially translates to “do there exist a Father and Mother such that the parents of Bill are Father and Mother?” Let’s try another question?

?- parents(harold, Father, Mother).

No

What has happened here? Clearly, Harold must have parents, but Prolog’s answer indicates otherwise. This highlights an important quirk about Prolog—a answer of “No” really means “not provable”. That is, we have not provided enough information for Prolog to answer our question. In particular, we haven’t said who Harold’s parents are.

We can also turn the question around and ask “who are Harold and Natalie’s children?”

?- parents(Child, harold, natalie).

Child = bill

At this point, Prolog has stopped and left the cursor next to the answer. If we simply press Enter now, like we did above, it will say “Yes” and give us a new prompt. However, we have another option—we can type a semi-colon, and Prolog tries to find another answer to the question:

?- parents(Child, harold, natalie).

Child = bill ;
Child = liane ;
Child = susan ;

No

Each time it finds a solution, we have the option of stopping or continuing. However, when we have exhausted all solutions, Prolog says “No” and gives us a new prompt.

Prolog also lets us combine questions with a comma, which is read “and”. We need this to ask things like “are there any other children of Greg’s parents?”

?- parents(greg, Father, Mother), parents(Sibling, Father, Mother).

Father = bill
Mother = joyce
Sibling = greg ;

Father = bill
Mother = joyce
Sibling = nick ;

No
The question asks “do there exist Father, Mother, and Sibling such that Greg’s parents are Father and Mother and Sibling’s parents are also Father and Mother?” This is not exactly what we want, since Sibling should be different from Greg. However, we can easily add this constraint to our question.

?- parents(greg, Father, Mother), parents(Sibling, Father, Mother),
   | Sibling \== greg.

Father = bill
Mother = joyce
Sibling = nick ;

No

This is a rather cumbersome and inflexible way of asking who someone’s siblings are. It would be much nicer if we had, for instance, a simple relation “sibling(S1, S2)” that held whenever S1 and S2 are siblings. Then we could simply say, for example:

?- sibling(greg, S).
S = nick ;
No
?- sibling(bill, S).
S = liane ;
S = susan ;
No

This requires about a fifth of the typing and also gives nice, clean answers without superfluous information about the parents. Implementing this sibling relation is really quite easy. In the file tree.prolog, we can add the following statement:

sibling(S1, S2) :-
parents(S1, Father, Mother),
parents(S2, Father, Mother),
S1 \== S2.

We read the :-) symbol as “if”. The relation we’re defining comes before the :-, while its definition comes after it. Note how the definition resembles what we originally typed at the prompt, except that greg has been generalized to S1.

An important difference between functional and logic programming is that variables in logic programs are not merely for input or output. Through unification, they can propagate constraints either forward or backward, as necessary. For example, we can ask sibling(greg, X) or sibling(X, greg), and either way, we’ll get the same answer. We can even ask, simply, sibling(X, Y) and get all pairs of siblings.

1 Automata

To represent a family tree in Prolog, we entered one simple statement, or fact, for each edge of interest in the tree. To represent automata, we can take a similar approach, providing one statement for each transition in the automaton. For example, we can build an automaton that accepts all strings with even numbers of a’s and b’s. This requires four states and eight transitions. We can encode it in Prolog as follows:
trans(start, a, a).
trans(start, b, b).
trans(a, a, start).
trans(a, b, ab).
trans(b, a, ab).
trans(b, b, start).
trans(ab, b, a).
trans(ab, a, b).

We also need to declare the start state as the accepting state:
accepting(start).

Finally, we need a rule to run the automaton. Given a current state and a list of input symbols, it will tell us the final state of the automaton:
finalState([], CurrentState, CurrentState).

finalState([Sym|Syms], CurrentState, FinalState) :-
    trans(CurrentState, Sym, NextState),
    finalState(Syms, NextState, FinalState).

The first rule just says that, when there’s no more input, we halt in the current state. The second rule says that, if there is at least one more symbol of input, we should take the transition for that symbol and then handle the rest of the input recursively.

Now we can determine whether a given string will be accepted by the automaton or not:
?- finalState([a, b, b, a, b, a, b], start, Final), accepting(Final).
Final = start
Yes

?- finalState([a, b, b, a, a, b, b], start, Final), accepting(Final).
No

?- finalState([a, b, b, a, a, b, b], start, Final).
Final = ab
Yes

?- finalState([a, b, c], start, Final).
No

Note the distinction between the second and fourth tests. In the second, the automaton processes the whole input, ending in a non-accepting state. In the fourth, it cannot process the input, since there are no transitions defined for the symbol c. We can exploit this behavior to simplify our descriptions of automata—when a set of symbols result in failure, we can simply leave out transitions for them.

We can also use this description of the automaton to generate strings that it accepts:
?- length(Input, 6), finalState(Input, start, Final), accepting(Final).
Input = [a, a, a, a, a]
Final = start ;
Input = [a, a, a, a, b, b]
Final = start ;

...