Writing and running simple Prolog programs

This lecture will cover writing and querying facts without and with unbound variables, and show how rules are a way of querying more than one fact at a time.
A question from last week

Why can’t Prolog have the ++ operator, for instance:

A_Variable++

Answer:

A_Variable ++

is the equivalent of:

A_Variable is A_Variable + 1

Prolog doesn’t allow variables to be reassigned.
Last time ...

Prolog’s built-in predicates and user-defined predicates look the same and behave the same.

Every predicate in Prolog returns true (yes) or fail (no).

Variables can’t be reassigned - it’s not logical.

\[
\text{Var is Var +1}
\]

is as logical as saying:

\[
1 \text{ is } 1 + 1.
\]
Last time ...

For your rules to communicate with other rules, you have to:

– send information in through the parameter list;
– send information back out through the parameter list.
Last time ...

When there is a choice of rules,
– Prolog tries the first and, if it works, returns “true” and carries on

If the rule fails,
– it tries the next, and the next and the next, etc, until:
– either it finds a true rule or it fails (catastrophically)
Today

This lecture will cover:
- revise unification
- introduce facts
- show that Prolog uses unification when searching for clauses
- writing rules that use facts
- how Prolog works – its “hidden” data structure
Tutorial Sheet 1 had a number of matching and unification questions.

Prolog has several predicates that match or compare two terms:

- \(==/2\) e.g. \(1 == 1\)
- \(\backslash==/2\) e.g. \(1 \backslash== 2\)
- \(=:=/2\) e.g. \(5 =:= 2 + 3\)
- \(=\backslash=/2\) e.g. \(7 =\backslash= 16 / 2\)
- \(>=/2\) e.g. \(5 >= 3\)

etc
Unification revisited - 2

None of these predicates assigns values to a variable:
- `=/2`, `\=/2`, `:=/=2`, `\=\=/2`, `>=/2`, etc

Only two predicates assign values:

Arithmetic only:
- `is/2` e.g. `Var1 is 3 * 4`

Unification:
- `=/2` e.g. `Var2 = an_atom`
When is unification true?

Unifying a variable with a literal:

\[ \text{Var1} = \text{an_atom}. \]
\[ \text{Var2} = 300.045. \]
\[ \text{Var3} = \text{an_object(an_atom, 300.045)}. \]
Unification revisited - 3

When is unification true?

Unifying a variable with a literal:

\[
\text{an\_atom} = \text{Var1}.
300.045 = \text{Var2}.
\text{an\_object}(\text{an\_atom}, 300.045) = \text{Var3}.
\]

The “empty” variable does not have to be on the left-hand side.
When is unification true?

Unifying a variable with another variable:

\[
| \ ?- \ Var1 = Var2. \\
Var2 = Var1 \ ; \\
no
\]

All unification really does is to make variables share memory locations.

Prolog show the variable binding - the “contents” of the variable.

The “semi-colon” in Prolog means “or”. We’re saying “Or is there another solution...?”

When is unification true?

Unifying a variable with another variable:

\[
| \ ?- \ Var1 = Var2. \\
Var2 = Var1 \ ; \\
no
\]

All unification really does is to make variables share memory locations.

Prolog show the variable binding - the “contents” of the variable.

The “semi-colon” in Prolog means “or”. We’re saying “Or is there another solution...?”
Unification revisited - 4

We can run a conjunction of unifications:
Writing facts - 1

In the last lecture, we saw rules such as:

define_facts(display_balance(bank_account(No, Name, Balance)) :-
    write(Name),
    write(' account no: '),
    write(No),
    nl,
    write('Your balance is: '),
    write(Balance),
    nl.)
Writing facts - 2

Facts are rules without a body:

\[
\text{likes}(\text{max, julia}). \\
\text{likes}(\text{max, amabel}).
\]

They are always true, so we could also write them as:

\[
\text{likes}(\text{max, julia}) :- \\
\quad \text{true}. \\
\text{likes}(\text{max, amabel}) :- \\
\quad \text{true}.
\]
Queries with these facts
What facts represent

Facts with arguments are used to describe relationships between arguments.

E.g.

lives_in(max, london).
likes(max, amabel).
child(charles, amy, brian).
price(template, 3, 4.75).
assembly(arm, joint(ball,3)).
An insight

Prolog matches our query with facts using unification.

Prolog uses unification wherever it has to match anything with anything.
Consolidation moment

Facts are rules that are always true.

Prolog will attempt to return every solution – in the order that they occur in the program.

Prolog uses unification to match queries with rule heads and facts.
Jealousy: the jealous/2 rule

This is a simple program:

```
jealous(Jealous, Victim) :-
    likes(Person, Jealous),
    likes(Person, Victim).

likes(max, julia).
likes(max, amabel).
```

06-25433 – Logic Programming
Jealousy: the jealous/2 rule
States of the stack - 1

Step 1 - add to stack

jealous(julia, Victim) :-
likes(Person, julia),
likes(Person, Victim).

Step 2 - match first subgoal

jealous(julia, Victim) :-
  likes(max, julia),
  likes(max, Victim).
States of the stack - 2

Step 3 - match second subgoal

jealous(julia, julia) :-
    likes(max, julia),
    likes(max, julia).

jealous(julia, amabel) :-
    likes(max, julia),
    likes(max, amabel).
States of the stack - 3

Step 4 - report success to the user

\[ \text{jealous(julia, Who).} \]
\[ \text{Who = julia ;} \]

\[
\text{jealous(julia, amabel) :-}
\text{likes(max, julia),}
\text{likes(max, amabel).}
\]
States of the stack - 4

Step 5 - report success to the user

\[
| \ ?- \ jealous(julia, \ Who). \\
\text{Who} = \ julia \ ; \\
\text{Who} = \ amabel \ ; \\
\]

2 - Writing and running simple Prolog programs
States of the stack - 5

Step 6 - report no more solutions

| ?- jealous(julia, Who).
Who = julia ;
Who = amabel ;
no
Consolidation moment

Prolog will attempt to find all solutions – if you ask.

Finding multiple solutions is based on an internal stack – which you’ll never see but you’ll see the effects.
Some terminology

jealous(Jealous, Victim) :-
    likes(Person, Jealous),
    likes(Person, Victim).

likes(max, julia).
likes(max, amabel).
Some terminology

jealous(Jealous, Victim) :-
    likes(Person, Jealous),
    likes(Person, Victim).

likes(max, julia).
likes(max, amabel).
Some terminology

jealous(Jealous, Victim) :-
    likes(Person, Jealous),
    likes(Person, Victim).

likes(max, julia).
likes(max, amabel).

1 rule
2 facts

2 - Writing and running simple Prolog programs
Suppose we define our friend as being either:

- an *immediate* friend of ourselves

*or*

- a *friend of a friend*
Friendship - 2

We could code this as:

```prolog
friend_of(max, julia).
friend_of(max, amabel).
friend_of(amabel, richard).  % etc
```

1

```prolog
friend(Pers, Friend) :-
    friend_of(Pers, Friend).
```

2

```prolog
friend(Pers, Friend) :-
    friend_of(Pers, Inter),
    friend_of(Inter, Friend).
```
A question

How many subgoals would the longest friend/2 rule have to have?

or, to put it another way:

What is the maximum number of friends you could have?