COSC343: Introduction to Prolog

Lecture 4

In today's lecture

- ♢ Prolog and AI research
- ♢ Clauses and queries
- ♢ Facts and rules
- ♢ Constants and variables
- ♢ Lists and unification
- ♢ Prolog and search

Prolog and symbolic AI

'Classical' AI is basically all about search.

- Early AI researchers (e.g. Newell and Simon) claimed that any task requiring intelligence can be solved using clever look-ahead search strategies.
  - You formulate your problem as a state space graph.
  - Then you search it systematically to look for a goal state.

Prolog is a language which is designed for (symbolic) AI tasks.

- In Prolog, every program is formulated as a state space and a goal state.
- Prolog comes with an inbuilt tree-search program.
- This makes Prolog very different from other programming languages!

SWI Prolog

Russell and Norvig's book isn't tied to any particular programming language.
But in 343 we'll be using Prolog for the first part of the course.

- To run Prolog on the Linux lab machines, just type `pl`.
- You'll get a prompt like this:
  `?-`
- To exit Prolog, type `halt.`
  (N.B. You need the fullstop!)

The variety of Prolog we're using is called SWI Prolog.

- There are pointers to manuals and other information on the course web-page.

Prolog syntax: atoms

The central construct in Prolog is the **atom**.

- An atom consists of a **predicate** and zero or more arguments (called **terms**). For instance:

```
male(charlie)
child_of(charlie, harry)
charlie_has_big_ears
```

An atom is basically Prolog's representation of a *fact* about the world.

The number of arguments a predicate takes is called its arity.

Prolog syntax: clauses and facts

Prolog operates with a database of **clauses**.

- The simplest kind of clause is called a **fact**.
- A fact is just an atom, followed by a fullstop.
- You create a database of clauses simply by editing a text file. (They conventionally take the suffix `.pl`.)

For instance: here's a simple database, which we could save as `my_db.pl`:

```
male(charlie).
child_of(charlie,harry).
```

Programming in Prolog

To program in Prolog:

- You create a database offline, using a text editor.
- You start up Prolog, and get a prompt.
- Then you load the database:

```
?- consult("my\_db.pl").
```

- After it's been loaded, you can type queries in at the prompt, and Prolog will return results for these queries. For instance:

```
?- male(charlie).
Yes
?-
```

- Syntactically, a query looks just like a fact. But it's interpreted as a question.

How Prolog responds to a query

Let's say we load `my_db.pl` into Prolog:

```
male(charlie).
child_of(charlie,harry).
```

When we ask a query, Prolog runs through the facts in the database in order, trying to match it to one of them.

- If a match is found, Prolog replies with `Yes`:

```
?- male(charlie).
Yes
?-
```

- If no match is found, Prolog replies with `No`:

```
?- female(queen_victoria).
No
?-
```

Constants and variables

The terms (arguments) in a predicate can be constants or variables.

- Constants are `lower_case`
- Variables are `Upper_case`.

If variables are used in a query, then Prolog is allowed to match facts in the database by unifying or binding variables in the query with constants (or variables) in the database.

```
?- male(X).
X = charlie
Yes
?-
```

To solve this query, Prolog has bound the variable `X` to the value `charlie`.

Prolog's search strategy

Let's extend the database a bit:

```
child_of(liz, charlie).
child_of(liz, anne).
child_of(liz, andrew).
child_of(charlie, harry).
child_of(charlie, will).
child_of(anne, zara).
```

Prolog searches the database of clauses in order (first-to-last), so the first clause it matches will be the first one entered in the database.

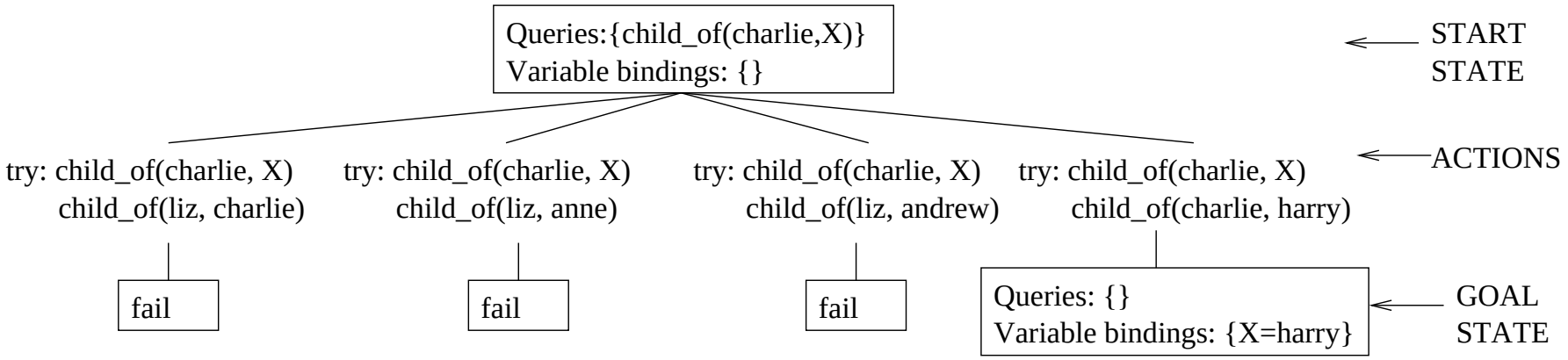
```
?- child_of(charlie, X).

X = harry
```

Visualising the search

Prolog basically executes a kind of tree search.

- Each of the system's actions is an attempt to match a query with one of the database clauses.
- The query plus the database define a set of possible states.
  - Each state reached by a *successful* match consists of a list of unresolved queries and a set of variable bindings.
  - An *unsuccessful* match results in a special state called `fail`.
- The goal state is one where the set of unresolved queries is empty.



Asking for other solutions

If Prolog finds a solution to a query containing variables, it asks the user if further solutions (involving different variable bindings) should be sought.

- If the user hits `<return>`, no further solutions are sought.
- If (s)he hits `;`, additional solutions are sought.

```
?- child_of(charlie, X).

X = harry ;
X = will ;

No
```

Prolog implements this by pretending that the goal state was a `fail` state, and continuing with its search.

Prolog rules

Our simple Prolog database just had facts in it.
A more complex kind of Prolog clause is a rule.

A rule has

- a head (a single Prolog atom),
- then the 'if' symbol (`:-`),
- then a body (a comma-separated list of atoms),
- then a fullstop.

For example:

```
loves(N1, N2) :-
        child_of(N2, N1).
```

(Read this as: 'N1 loves N2 if N2 is a child of N1'.)

Query-matching with rules

When a query is made, Prolog searches the clauses in order.

- If the clause is a fact, Prolog tries to match the query to it directly.

- If the clause is a rule, Prolog tries to match the query to the rule's *head*. If the head matches, then the result state is defined as follows:
  - The query matching the head of the rule is deleted from the list of queries.
  - All the terms in the *body* of the rule become queries themselves, and are added to the list.

Rules thus introduce searches of depth greater than 1.

> Note: new queries are added to the *front* of the list of queries.
> So Prolog implements a depth-first search.

An example

Consider this simple database:

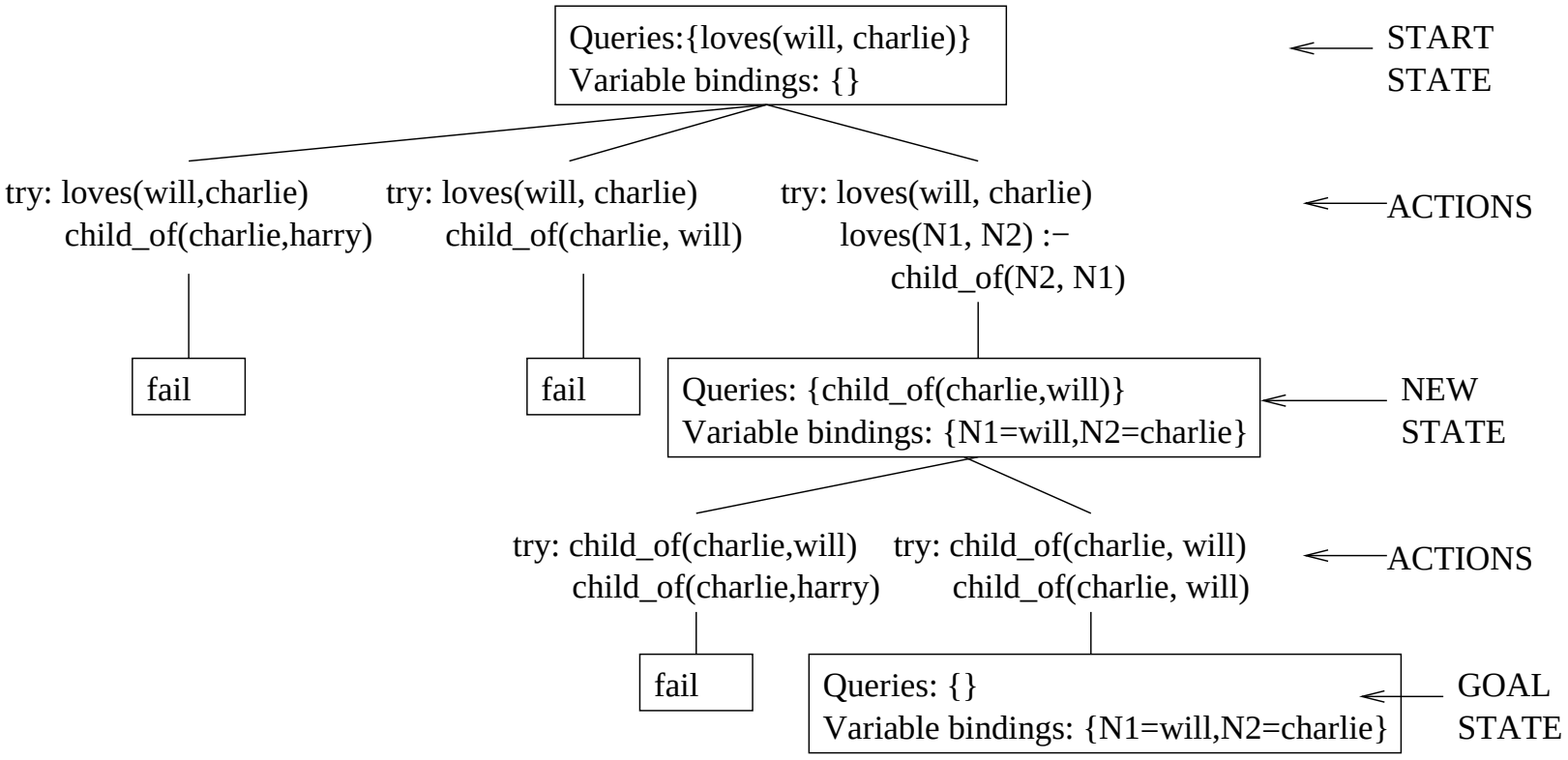
```
child_of(charlie, harry).
child_of(charlie, will).
loves(N1, N2) :-
        child_of(N2, N1).
```

And the query `loves(will, charlie).`

- Prolog runs through the clauses in order, trying to match each one.
- The first two clauses fail directly.
- The head of the third clause matches, if we bind `N1` to `charlie` and `N2` to `will`.
- We now generate a new sub-query to test:
  `child_of(charlie, will).`
- We test *this* query against each clause in the database, left-to-right. And this succeeds.

Visualising the search

```
child_of(charlie, harry).
child_of(charlie, will).
loves(N1, N2) :-
        child_of(N2, N1).          ?- loves(will, charlie).
```



Tracing

Prolog has a trace facility, to display its reasoning process.

Here's a trace of the previous query.

```
?- trace, loves(will,charlie).
   Call: (8) loves(will, charlie) ?
   Call: (9) child_of(charlie, will) ?
   Exit: (9) child_of(charlie, will) ?
   Exit: (8) loves(will, charlie) ?
```

Note:

- The numbers tell us how deep in the search Prolog is. (1–7 are part of SWI's user interface.)
- The default trace doesn't report intermediary 'fail' nodes.

Recursion in Prolog

Prolog rules can be *recursive*.

Here's an example of a recursive rule for defining 'descendant of':

```
descendant_of(N1, N2) :-
        child_of(N1, N2).
descendant_of(N1, N2) :-
        child_of(N1, Nmid),
        descendant_of(Nmid, N2).
```

The first of these rules is the base case.
The second rule is the recursive case.

N.B. The base case always has to appear first!

Term unification

When Prolog attempts to match two atoms, their predicates must be identical, and their arguments have to **unify**.

We can test matches explicitly using the infix = operator:

```
?- happy(bill) = sad(bill).
No
```

We can also use = to test directly for term unification:

```
?- X = bill.
X = bill
```

Term unification

A variable can unify with any term, provided that it can be substituted *consistently* for that term throughout the predicate.

How will Prolog respond to the following queries?

```
?- foo(X, X) = foo(bar, bar).
```

```
?- foo(X, Y) = foo(bar, bar).
```

```
?- foo(X, X) = foo(bar, baz).
```

Complex terms

Terms don't need to be constants or variables: they can also be more complex expressions.

- A whole atom can be a term: e.g. `loves(child_of(john), mary)`

Prolog has a set of inbuilt **operators** (e.g. `:`, `-`, `\`), which allow the creation of arbitrarily complex terms (e.g. `a:b:c`, `a-b-X`).

How will Prolog respond to the following queries?

```
?- a/b/c = a:b:c.
```

```
?- a/b/c = a/b/X.
```

```
?- f(g(a/b), Y) = f(g(Z), p).
```

Lists in Prolog

One special form of complex expression is a list. For instance:

```
corgis_of(liz, [rover, fido]).
```

- Prolog represents the *empty list* using a special symbol, `[]`.
- Prolog represents a non-empty list as a binary structure `[`x`|`y`]`, where
  - x is any Prolog term (simple or complex),
  - y is a list (empty or non-empty).

  A list with one element is shown as `[`e`|[]]`.
  A list with two elements is shown as `[`e`|[`$e2$`|[]]]`.

To make things simple, there's a shorthand:
E.g. `[a]` is a shorthand for `[a|[]]`.
E.g. `[a, b]` is a shorthand for `[a|[b|[]]]`.

Term unification

We can confirm these shorthand representations by unifying:

```
?- [a|[]] = X.
X = [a]
Yes

?- [a|X] = [a].
X = []
Yes

?- [a|X] = [a, b, c].
X = [b, c]
Yes

?-
```

A predicate for testing list membership

There's an inbuilt predicate called `member/2` that works like this:

```
?- member(a, [b, a, c]).
Yes

?- member(a, [b, c, d]).
No
```

We can define our own version of this predicate. Note the recursion!

```
member*(X, [X|Rest]).              %base case

member*(X, [Y|Rest]) :-            %recursive case
        member*(X, Rest).
```