A logic programming trick

- A two-way translator in two lines of code:

```prolog
translate([],[]).
translate([Word|Words],[Mot|Mots]) :-
    dict(Word,Mot),
    translate(Words,Mots).
```

- (not counting the dictionary)
- What it does:

```prolog
| ?- translate([the,dog,chases,the,cat],Francais).
  Francais = [le,chien,chasse,le,chat].
| ?- translate(English,[le,rat,mange,le,fromage]).
  English = [the,rat,eats,the,cheese] ?
```

A simple Prolog program

- When is an item an element of a list?
- Axiom:

```prolog
element(X,[X|Xs]).
```

- Inference rule:

```prolog
element(X,[Y|Xs]) :- element(X,Xs).
```

- Query:

```prolog
| ?- element(1,[2,1,4]).
  true
```
Not just a functional program

- Give me lists that 1 is a member of:
  
  | ?- element(1,Xs).
  
  Xs = [1|_] ? ;
  Xs = [_,1|_] ? ;
  Xs = [_,_,1|_] ?

- No limit to the number of answers

Running it "backwards"

- Give me the elements of a given list
  
  element(X,[2,1,4]).

  X = 2 ? ;
  X = 1 ? ;
  X = 4 ?;
  no

Informal semantics

- No evaluation - proof search instead
- "Variables" are bound as a result of search
- A "program" is a set of clauses together with a query
- The meaning of a program is a set of proofs
- The "answer" is yes or no - a proof was found or not - together with bindings for the variables

Stranger Prolog programs

- Generate-and-test
- Example: do two lists have a nonempty intersection?

  intersect(Xs,Ys) :-
    element(X,Xs), element(X,Ys).

  | ?- intersect([1,3,5],[2,3,5]).
  
  true ? ;
  true ? ;
  no
Problem definition

- Can we place $n$ queens on an $n$-by-$n$ chessboard so that no queen attacks any other queen?
- Idea: represent a solution as some permutation of the numbers $1...n$, each one giving the row number to place the queen in column $n$.
- By construction no two queens are in the same row or column.
- Our program checks whether any two share a diagonal.

Example solution

- $[2,5,7,1,3,8,6,4]$
Checking for attacks

- If two queens are \( n \) columns apart, there is an attack if they are also \( n \) rows apart.
- To do a safety check, check that the first queen doesn't attack any of the rest, and the second queen doesn't attack any of the rest, ....

Safety check

- \( \text{notattack}(X,1,L) \)
- \( \text{notattack}(2,1,[5,7,1,3,8,6,4]) \)

Defining safety

\[
\begin{align*}
\text{notattack}(X,N,[ ]) & . \\
\text{notattack}(X,N,[Y|Ys]) & : - \\
& X =\leq Y+N,
& X =\leq Y-N,
& N1 \text{ is } N+1,
& \text{notattack}(X,N1,Ys).
\end{align*}
\]

Generating candidates

- Generate the numbers 1..\( n \)
  \[
  \begin{align*}
  \text{range}(N,N,[N]) & . \\
  \text{range}(N,M,[N|Ns]) & : - M>N, \quad N1 \text{ is } N+1,
  \quad \text{range}(N1,M,Ns).
  \end{align*}
  \]
- Generate a permutation
  \[
  \begin{align*}
  \text{perm}([ ],[ ]) & . \\
  \text{perm}([X|Xs],Ys) & : - \text{perm}(Xs,Zs),
  \quad \text{append}(Z1,Z2,Zs),
  \quad \text{append}(Z1,[X|Z2],Ys).
  \end{align*}
  \]
**Splitting with append**

?- append(Xs,Ys,[1,2,3]).

Xs = []
Ys = [1,2,3] ? ;

Xs = [1]
Ys = [2,3] ? ;

Xs = [1,2]
Ys = [3] ? ;

Xs = [1,2,3]
Ys = [] ? ;

(1 ms) no

---

**Generate-and-test**

queens(N,Qs) :- range(1,N,Ns),
perm(Ns,Qs),
safe(Qs).

safe([]).
safe([Q|Qs]) :- safe(Qs),
notattack(Q,1,Qs).

---

**What it does**

?- queens(8,Qs).

Qs = [5,2,6,1,7,4,8,3] ? ;

Qs = [6,3,5,7,1,4,2,8] ? ;

Qs = [6,4,7,1,3,5,2,8] ? ;

Qs = [3,6,2,7,5,1,8,4]

... and 88 other solutions

---

**Unification makes it work**

- Unification: given two terms $t_1$ and $t_2$, both potentially containing variables, can we find a substitution for those variables making $t_1$ and $t_2$ the same?
- e.g. unify $[X,3,4|Xs]$ and $[2,3,Y|Ys]$:
  - $\{ X:=2, \ Xs:=Ys, \ Y:=4 \}$
Unification in n-queens

- Unify queens(8,Qs) with queens(N,Qs)
- Result: { N:=8}
- New goal:
  range(1,8,Ns),
  perm(Ns,Qs),
  safe(Qs).

Backtracking makes it work

- A search tree

Backtracking trace

- Remember element testing:
  
  element(X,[X|Xs]).
  element(X,[Y|Xs]) :- element(X,Xs).

- Tracing it:

<table>
<thead>
<tr>
<th>?- element(X,[1,2,3]).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call: element(X,[1,2,3]) ?</td>
</tr>
<tr>
<td>Exit: element(1,[1,2,3]) ?</td>
</tr>
<tr>
<td>X = 1 ? ;</td>
</tr>
<tr>
<td>Call: element(X,[2,3]) ?</td>
</tr>
<tr>
<td>Exit: element(2,[2,3]) ?</td>
</tr>
<tr>
<td>Exit: element(2,[1,2,3]) ?</td>
</tr>
<tr>
<td>X = 2 ? ;</td>
</tr>
<tr>
<td>Call: element(X,[3]) ?</td>
</tr>
<tr>
<td>Exit: element(3,[3]) ?</td>
</tr>
<tr>
<td>Exit: element(3,[2,3]) ?</td>
</tr>
<tr>
<td>Exit: element(3,[1,2,3]) ?</td>
</tr>
<tr>
<td>X = 3 ? ;</td>
</tr>
</tbody>
</table>

Extensions

- Change the search order, e.g. to breadth-first
- Constraint solvers
- Function definitions
- Higher-order unification
- ...
Why learn logic programming?

- Expand your view of computation
- Acquire a powerful specialized tool
- Amaze and baffle your friends in 50 lines of code:

```
i am sure there are space aliens around.
how long have you been sure there are space aliens
around ?
since my mother went crazy.
can you tell me more about mother
i like to pull her hair.
does anyone else in your family like to pull her hair ?
my brother.
can you tell me more about brother
he is too weird.
please go on
i feel he is watching me.
do you often feel that way ?
```
CS301
Session 21

Agenda
- Semantics of Prolog
  - Logical view
  - Substitutions
  - Unification
  - Procedural view

Logical vs. procedural semantics
- Logical semantics extremely simple but it's an idealization of what actually happens
- It ignores effects of search order, e.g. nontermination
- Procedural semantics specifies search order
- Can also specify the behavior of the nonlogical constructs like cut

Logical semantics
- Judgment: the conjunction of goals is satisfiable using the set of clauses \( D \) and the substitution \( \theta \)
  \[ D \vdash \hat{\theta}g_1, \ldots, \hat{\theta}g_n \]
- Rule for conjunctions
  \[
  \frac{D \vdash \hat{\theta}g_1 \quad \ldots \quad D \vdash \hat{\theta}g_n}{D \vdash \hat{\theta}g_1, \ldots, \hat{\theta}g_n}
  \]
Logical semantics cont'd

- Rule for a single goal

\[ C \in D \quad C = G :- H_1, \ldots, H_m \]
\[ \hat{\theta}'(G) = \hat{\theta}g \]
\[ D \vdash \hat{\theta}'(H_1), \ldots, \hat{\theta}'(H_m) \]
\[ D \vdash \hat{\theta}g \]

- C is any clause in the database!

Substitutions

- Informally, think of a substitution as a function that maps logic variables to Prolog terms (which may contain logic variables)

- If \( \theta \) a substitution and \( t \) a term, write \( \theta t \) for the application of \( \theta \) to \( t \)

- but write \( \hat{\theta}g \) for the application to a goal \( g \)

- A substitution never affects a functor, predicate, or literal

Unification

- Unification plus variable renaming finds the pair of substitutions we need to match a goal to a clause head

- Why renaming? Consider:

  \[
  \text{member}(M,[1\mid\text{nil}])
  \]
  \[
  \text{member}(X,[X\mid M])
  \]

- We need to consider the two occurrences of \( M \) to be different variables.

Unification: two subtleties

- Unification finds a most general unifier! We're not interested in other substitutions.

- To be correct, unification must do an occurs check: the following should not unify:

  \[
  \text{foo}(X,[X\mid L])
  \]
  \[
  \text{foo}(Y,[\text{bar}(Y)\mid M])
  \]
Procedural semantics
- Specifies order of evaluation
- which clause is matched first?
- how does backtracking work?

Choosing a clause
- Given an atomic query \( g \) and a database \( D \), we attempt to satisfy \( g \) using the clauses of \( D \) in the order in which they appear.
- This yields nontermination in the following:

\[
\begin{align*}
\text{element}(X,[Y|Xs]) & :\text{e} \text{lement}(X,Xs). \\
\text{element}(X,[X|Xs]) & . \\
?- \text{element}(1,L).
\end{align*}
\]

Backtracking
- If we unify a goal with a clause \( C \), but fail to satisfy a subgoal, we return to the list of clauses and try to unify our goal with the next clause after \( C \).
- This causes nontermination in:

\[
\begin{align*}
\text{reach}(X,Y) & :\text{reach1}(X,Y). \\
\text{reach}(X,Y) & :\text{reach}(X,U), \text{reach}(U,Y). \\
\text{reach}(X,X). \\
?- \text{reach}(a,a).
\end{align*}
\]

Comparing the two
- The logical interpretation is "too powerful" - if there is any way to find a proof, it succeeds.
- The procedural interpretation reflects what can be easily, efficiently implemented, but is harder to understand.
- Note that many implementations omit the "occurs check" to speed up unification.
Exercise

- Small groups - do exercise 2 (a) and (b)