Prolog Lecture 3

- Symbolic evaluation of arithmetic
- Controlling backtracking: cut
- Negation
Symbolic Evaluation

Let's write some Prolog rules to evaluate symbolic arithmetic expressions such as \texttt{plus(1,mult(4,5))}

\begin{align*}
\text{eval}(\text{plus}(A,B),C) & :\quad \text{eval}(A,A1), \\
& \quad \text{eval}(B,B1), \\
& \quad C \text{ is } A1 + B1. \\
\text{eval}(\text{mult}(A,B),C) & :\quad \text{eval}(A,A1), \\
& \quad \text{eval}(B,B1), \\
& \quad C \text{ is } A1 \times B1. \\
\text{eval}(A,A). 
\end{align*}
Evaluation starts with the first matching clause

Q: How does Prolog evaluate:

\[
\text{eval(plus(1,mult(4,5)),Ans)}
\]

A: Step 1, see if the first matching clause is true

\[
\text{eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.}
\]

In this case the variable bindings are:
- \(A = 1, B = \text{mult}(4,5)\) and \(C = \text{Ans}\)
Next it looks at the body of the rule

The body of the clause with head
\[ \text{eval(plus}(A,B),C) \] and variable bindings
\[ A = 1, \ B = \text{mult}(4,5) \] and \[ C = \text{Ans} \] is:

\[
\begin{align*}
\text{eval}(1,A1), \\
\text{eval}(\text{mult}(4,5),B1), \\
\text{Ans} \text{ is } A1 + B1.
\end{align*}
\]

This is a conjunction: all parts must be true for the clause to be true
The body is checked term by term from left to right

First part of the body: eval(1,A1)

Fail because 1 does not unify with plus(A,B)

Fail because 1 does not unify with mult(A,B)

Try: eval(A,A).
Succeed: eval(1,A1) is true if A1 = 1
The body is checked term by term from left to right.

From previous slide, \texttt{eval(1,A1)} was provable, with the side-effect of binding: \texttt{A1=1}.

So continuing through the body (note \texttt{A1} is now bound):

\begin{verbatim}
  eval(1,1),
  eval(mult(4,5),B1),
  Ans is 1 + B1.
\end{verbatim}
The body is checked term by term from left to right

So \( \text{eval(mult(4,5),B1)} \) will bind \( B1=20 \):

\[
\begin{align*}
\text{eval(1,1),} \\
\text{eval(mult(4,5),20),} \\
\text{Ans is 1 + 20.}
\end{align*}
\]
The body is checked term by term from left to right

Ans will be bound to 21, after “is” does its job.

eval(1,1),
eval(mult(4,5),20),
21 is 1 + 20.
Be sure that you understand why the second eval/3 clause does not appear in this choice point.
eval(plus(1,mult(4,5)),Ans)

\[
eval(plus(1,mult(4,5)),\text{Ans}) ::
\begin{align*}
&\text{eval}(1,T1), \quad \text{eval}(\text{mult}(4,5),T2), \quad \text{Ans is } T1 + T2.
\end{align*}
\]

eval(plus(A,B),C) :-
\[
\text{eval}(A,A1), \quad \text{eval}(B,B1), \quad \text{C is } A1 + B1.
\]

eval(mult(A,B),C) :-
\[
\text{eval}(A,A1), \quad \text{eval}(B,B1), \quad \text{C is } A1 \times B1.
\]

eval(A,A).

eval(1,1).
eval(plus(A,B), C) :-

eval(plus(1, mult(4,5)), Ans) :-
    eval(1, T1), eval(mult(4,5), T2), Ans is T1 + T2.

eval(mult(A,B), C) :-

eval(mult(4,5), T2) :-
    eval(4, T3), eval(5, T4), T2 is T3 * T4.

eval(A, A).

eval(plus(A, B), C) :-

eval(plus(1, mult(4,5)), Ans) :-
    eval(1, T1), eval(mult(4,5), T2), Ans is T1 + T2.

eval(A, A).

eval(1, 1).
eval(plus(1, mult(4, 5)), Ans) :-
    eval(1, T1), eval(mult(4, 5), T2), Ans is T1 + T2.

eval(mult(A, B), C) :-

L3-12
eval(A,A).
eval(4,4).
eval(plus(1,mult(4,5)),Ans) :-
    eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.
eval(mult(A,B),C) :-
eval(plus(A,B),C) :-
eval(A,A).
eval(1,1).
eval(A,A).
eval(5,5).
eval(A,A).
eval(A,A).
eval(1,1).
eval(A,A).
eval(mult(4,5),T2) :- eval(4,T3), eval(5,T4), T2 is T3 * T4.
eval(A,A).
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.
eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.
eval(plus(1,mult(4,5)),Ans) :-
    eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

20 is 5 * 4.

21 is 1 + 20.
What happens if we use backtracking and ask Prolog for the next solution?
eval(plus(1, mult(4, 5)), Ans) :-
  eval(1, T1), eval(mult(4, 5), T2), Ans is T1 + T2.

20 is 5 * 4.

21 is 1 + 20.
eval(plus(1,mult(4,5)),Ans) :-
  eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(plus(1,mult(4,5)),Ans) :-
  eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

20 is 5 * 4.

eval(plus(A,B),C) :-

eval(plus(A,B),C) :-

eval(mult(A,B),C) :-

eval(mult(A,B),C) :-
eval(plus(1,mult(4,5)),Ans) :-
    eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(mult(A,B),C) :-

eval(plus(A,B),C) :-

eval(A,A).
eval(plus(1, mult(4, 5)), Ans) :-
    eval(1, T1), eval(mult(4, 5), T2), Ans is T1 + T2.

eval(A, A).

eval(mult(A, B), C) :-

eval(mult(4, 5), T2) :-
    eval(4, T3), eval(5, T4), T2 is T3 * T4.

eval(A, A).

eval(plus(A, B), C) :-

eval(A, A).

eval(4, 4).

eval(A, A).

eval(plus(1,mult(4,5)),Ans) :-
    eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(mult(4,5),T2) :-
    eval(4,T3), eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(plus(1, mult(4,5)), Ans) :-
  eval(1, T1), eval(mult(4,5), T2), Ans is T1 + T2.

eval(plus(A, B), C) :-

eval(mult(A, B), C) :-
eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

Ouch... “is” can't handle the mult(4,5) term!
(a) Eliminate spurious solutions by making your clauses orthogonal

Need to eliminate the (unwanted) choice point

A way to do this: make sure only one clause matches: \texttt{eval(A,A)} becomes \texttt{eval(gnd(A),A)}.

\begin{verbatim}
 eval(plus(A,B),C) :- eval(A,A1),
                    eval(B,B1),
                    C is A1 + B1.
 eval(mult(A,B),C) :- eval(A,A1),
                    eval(B,B1),
                    C is A1 * B1.
 eval(gnd(A),A).
\end{verbatim}
(b) Eliminate spurious solutions by explicitly discarding choice points

Alternatively we can tell Prolog to commit to its first choice and discard the choice point \( \text{p114} \)

We do this with the cut operator. Written: !

```
 eval(plus(A,B),C) :- !,eval(A,A1), eval(B,B1), C is A1 + B1.
 eval(A,A).
```
eval(A,A).

eval(4,4).

20 is 5 * 4.

eval(plus(1,mult(4,5)),Ans) :- !,eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(plus(1,mult(4,5)),Ans) :- !,eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

These choices are eliminated

These choices are eliminated

Cuts

eval(plus(A,B),C) :- !,eval(A,A1), eval(B,B1), C is A1 + B1.


eval(A,A).

eval(1,1).


eval(mult(4,5),T2) :- !,eval(4,T3), eval(5,T4), T2 is T3 * T4.

eval(A,A).

eval(A,A).

eval(5,5).

eval(A,A).

eval(A,A).

These choices are eliminated

21 is 1 + 20.
Cutting out choice

Whenever Prolog evaluates a cut it discards all choice points back to the parent clause

An example:

```prolog
a(1).
c(A, B, C) :- a(A), d(B, C).
a(2).
c(A, B, C) :- b(A), d(B, C).
a(3).
d(B, C) :- a(B), !, a(C).
b(apple).
d(B, _) :- b(B).
b(orange).
```
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
c(A, B, C) :- a(A), d(B, C).
c(A, B, C) :- b(A), d(B, C).
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A, B, C) :- a(A), d(B, C).
c(A, B, C) :- b(A), d(B, C).
d(B, C) :- a(B), !, a(C).
d(B, _) :- b(B).
c(A,B,C) 

\[
c(A,B,C) : -a(A), d(B,C).
\]

\[
c(A,B,C) : -b(A), d(B,C).
\]
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).
d(B,C):-b(B).
d(B,_):-b(B).
a(1).
a(2).
a(3).
\begin{align*}
&c(A,B,C) :- a(A), d(B,C). \\
&c(A,B,C) :- b(A), d(B,C). \\
&d(B,C) :- a(B), !, a(C). \\
&d(B,C) :- b(B). \\
\end{align*}
c(A, B, C)

\[ c(A, B, C) : \neg a(A), d(B, C). \]
\[ c(A, B, C) : \neg b(A), d(B, C). \]

\[ a(1). \]
\[ a(2). \]
\[ a(3). \]

\[ b(apple). \]
\[ b(orange). \]

\[ c(A, B, C) : a(A), d(B, C). \]
\[ c(A, B, C) : b(A), d(B, C). \]
\[ d(B, C) : a(B), !, a(C). \]
\[ d(B, C) : \neg a(B), !, a(C). \]
\[ d(B, C) : b(B). \]
\[ d(B, _) : \neg b(B). \]
c(A,B,C)

c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).

da(B,C):-a(B),!,a(C).
da(B, _):-b(B).

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

Backtrack twice
c(A,B,C)

c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).

d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).
d(B,C):-a(B),!,a(C).

Backtrack three times
c(A,B,C):=a(A),d(B,C).
c(A,B,C):=b(A),d(B,C).
c(A,B,C):=a(A),d(B,C).
c(A,B,C):=b(A),d(B,C).
d(B,C):=a(B),!,a(C).
d(B,C):=a(B),!,a(C).
d(B,C):=a(B),!,a(C).
d(B,C):=a(B),!,a(C).
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):=a(A),d(B,C).
c(A,B,C):=b(A),d(B,C).
c(A,B,C):=b(A),d(B,C).
c(A,B,C):=b(A),d(B,C).
d(B,C):=a(B),!,a(C).
d(B,C):=a(B),!,a(C).
d(B,C):=a(B),!,a(C).
d(B,C):=a(B),!,a(C).
First a/1 has other solutions
Can try to derive d/2 afresh...
Cut can change the logical meaning of your program

\[
p \iff (a \land b) \lor c
\]

\[
p \iff (a \land b) \lor (\neg a \land c)
\]

This is a red cut – **DANGER!**  (p128)
Cut can be used for efficiency reasons

\[
\text{split}([],[],[]).
\]
\[
\text{split}([H|T],[H|L],R) :- H < 5, \text{split}(T,L,R).
\]
\[
\text{split}([H|T],L,[H|R]) :- H \geq 5, \text{split}(T,L,R).
\]

If the second clause succeeds the third cannot
- we don't need to keep a choice point
- yet the interpreter cannot infer this on its own
Cut can be used for efficiency reasons

Add a cut to make the orthogonality explicit

- This is a green cut – it just helps program execution go faster

```prolog
split([],[],[]).
split([H|T],[H|L],R) :- H < 5,!, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```
We could go one step further at the expense of readability

The comparison in the third clause is no longer necessary
- but now each clause does not stand on its own
- stylistic preference – I avoid doing this
Programmers new to Prolog often cause determinism errors

Check that your predicates return the correct number of answers!
- e.g. providing a correct answer multiple times is likely to cause bugs that are difficult to find

Below is a predicate you can use in debugging
- (Note that findall is not discussed in lectures)

\[
\text{numsol}(	ext{Predicate}, \text{NumberOfSolutions}) :- \\
\text{findall}(	ext{dummy}, \text{Predicate}, \text{AnsList}), \\
\text{length}(	ext{AnsList}, \text{NumberOfSolutions}).
\]
Testing using the numsol predicate

A warning will be generated about mergesplat/3.
- What is wrong with the mergesplat/3 predicate?

```prolog
mergesplat([],[],[]).
mergesplat(A, [], A).
mergesplat([], B, B).
mergesplat([A|As], [B|Bs], [A,B|Rest]):-
    mergesplat(As, Bs, Rest).

:-numsol(merge([1,2],[a,b],_),1).
:-numsol(mergesplat([1,2],[a,b],_),1).
```
Cut gives us more expressive power

```prolog
isDifferent(A,A) :- !, fail.
isDifferent(_,_).
```

isDifferent(A,B) is true iff A and B do not unify

Questions that you should be able to answer:
- Is this a red or a green cut?
- How can you define the fail/0 predicate?
Using cut, we can implement “not” 
(Negation by failure)

\[
\text{not}(A) :- A,!,\text{fail}.
\text{not}(_).
\]

\text{not}(A) \text{ is true if } A \text{ cannot be shown to be true}
- This is \text{negation by failure} \ (p124)

\text{Negation by failure is based on the closed world assumption:} \ (p129)

\text{Everything that is true in the “world” is stated (or can be derived from) the clauses in the program}
Negation Example

good_food(theWrestlers).
good_food(theCambridgeLodge).
expensive(theCambridgeLodge).

bargain(R) :- good_food(R),
            not(expensive(R)).

we can ask:
- bargain(R)

and Prolog replies:
- R = theWrestlers
Negation Gotcha!

we can ask the same query:
- bargain(R)

and Prolog replies:
- no

Clause body terms have been swapped around!
Why?

Prolog first tries to find an R such that `expensive(R)` is true.
- therefore `not(expensive(R))` will fail if there are any expensive restaurants

```prolog
good_food(theWrestlers).
good_food(theCambridgeLodge).
expensive(theCambridgeLodge).

bargain(R) :- not(expensive(R)),
good_food(R).
```
We sometimes identify the way to use parameters of a rule

Prolog's non-logical properties can make it important whether or not an argument to a predicate is bound

% indicates a comment to the end of that line

% this comment in some hypothetical code is % describing how to query myrule(+A,+B,-C,-D)

The convention for comments about rule parameters:
- +X is a ground term
- -X is a variable term
- ?X means it does not matter

Query “myrule” with two ground (input) terms A and B and two variable (output) terms C and D
Prolog variables and quantifiers

When R is not bound, quantifiers need attention

expensive(R)
  - “There exists an R that is expensive”.

not(expensive(R))
  - “There does not exist an R that is expensive”.
  - In other words, “for all R, not expensive(R)”.
Databases

Information can be stored as tuples in Prolog's internal database

```
tName(dme26,'David Eyers').
tName(awm22,'Andrew Moore').

tGrade(dme26,'IA',2.1).
tGrade(dme26,'IB',1).
tGrade(dme26,'II',1).
tGrade(awm22,'IA',2.1).
tGrade(awm22,'IB',1).
tGrade(awm22,'II',1).
```
We can now write a program to find all names:

\[ \text{qName}(N) :- \text{tName}(\_, N). \]

Or a program to find the full name and all grades for dme26.

\[ \text{qGrades}(F, C, G) :- \text{tName}(I, F), \text{tGrade}(I, C, G). \]

Further exercises are in the problem sheet...