Lecture 11 (Oct 21)

Lecture outline:

- more on negation
- anonymous variables
- cut

Some peculiarities of negation-as-failure

Consider the following Prolog program (`beer.pl`):

```
likes(anton, B) :- beer(B), \+ cheap(B).
beer(lakeport).
cheap(lakeport).
beer(creemore).
```

Running the query `?- likes(anton, X).` gives the answer `X = creemore`, as expected. Note, that from the logical point of view, the order of subgoal in the first clause should not affect the answer, in other words, rewriting the first clause as

```
likes(anton, B) :- \+ cheap(B), beer(B).
```

should produce the exact same result. However, due to the fact that negation in Prolog is implemented as negation-by-failure, the modified program behaves differently:

```
?- likes(anton, X).
No
```

Another interesting point: double-negation is not the same as absence of negation. Consider the following queries:

```
?- cheap(X).
X = lakeport ;
No

?- \+ \+ cheap(X).
X = _G180 ;
No
```

The reason that X is not instantiated in the second query is that all bindings performed during the negation-as-failure check are lost.

Anonymous variables

Let's say we'd like to modify the `beer.pl` program so that the first clause would apply to anybody, and not just to `anton`, i.e.

```
likes(X, B) :- beer(B), \+ cheap(B).
```

Note that the variable X is not used anywhere in the clause body – Prolog will print the warning that X is a "singleton" variable. In this case we know that we don't really need X, and so instead of X we can put _, like this:

```
likes(_, B) :- beer(B), \+ cheap(B).
```

Every time Prolog sees _ it generates a *new* internal variable, whose bindings will not be displayed. It is very important to understand that every occurrence of _ is replaced by a new variable. For example, the clause

```
p(_) :- q(_), r(_).
```

is equivalent to

```
p(X) :- q(Y), r(Z).
```

and **not** to

```
p(X) :- q(X), r(X).
```

Consider, again, our beer example:

```
?- beer(_), \+ cheap(_).
No
```

but

```
?- beer(X), \+ cheap(X).
X = creemore
Yes
```

Cut

The special predicate *cut*, written as `!`, allows to control the way the search space is constructed. Cut in a goal always succeeds, but it has a side effect – it prunes off a chunk of the search tree. Reasons to want to do this could be many: performance considerations, determinism, and others.
For starters, let us consider cut in goals. Take for example the following goal, which checks whether a given list L contains to distinct numbers:

```
?- member(X,L), number(X), member(Y,L), number(Y), X \= Y.
```

For example,

```
?-  member(X,[1,a,b,c,3]), number(X), member(Y,[1,a,b,c,3]), number(Y), X \= Y.
X = 1
Y = 3 ;

X = 3
Y = 1 ;
No
```

Now, consider the situation when the list does not have two integers:

```
?-  member(X,[1,a,b,c]), number(X), member(Y,[1,a,b,c]), number(Y), X \= Y.
No
```

It surely fails, but after the subgoals

```
member(Y,[1,a,b,c]), number(Y), X \= Y.
```

fail, Prolog does a lot of work: it binds X to the rest of the members of the list, and does the `number(X)` check for all of them. This is clearly unnecessary because the failure of the three subgoals below indicates that there are no other suitable number in the list, and so there's no point to search further. Inserting cut as follows:

```
?-  member(X,[1,a,b,c]), number(X), !, member(Y,[1,a,b,c]), number(Y), X \= Y.
No
```

tells Prolog to stop searching once the backtracking attempts to "cross" the cut. So, cut in goals, acts as a "barrier" – once the search crosses the cut, the backtracking cannot cross it back. Notice the behavior of the query with cut for the list with two numbers:

```
?-  member(X,[1,a,b,c,3]), number(X), !, member(Y,[1,a,b,c,3]), number(Y), X \= Y.
X = 1
Y = 3 ;
No
```

Cut in program clauses as a bit more tricky. Consider the following program:

```
a :- b, c.
a :- f.
b :- d, !, e.
b :- g.
d.
f.
```

Execution of the top-level query `?- a.` produces the following sequence of goals:

```
:- a.
:- b, c.
:- d, !, e, c.
:- !, e, c.
```

Now, cut always succeeds, so we get

```
:- e, c.
fail.
```

At this point, usual Prolog backtracking would try `b :- g.`, however the presence of cut forces Prolog to backtrack to `:- a.`, and try `a :- f.` instead. The rule that allows to determine the backtrack point whenever the cut is encountered is as follows:

1. determine which goal caused the selection of clause with cut (`:- b, c.` in our case.
2. backtrack to its parent (`:- a.` in our case), and try another alternative (the clause `a :- f.` in our case).

If the goal in 1. is the top-level goal (i.e. it does not have a parent), then simply fail. For example, for the program

```
a :- b, !, c.
a :- f.
b.
f.
```

the query `?- a.` fails.

When and how to use cut.

One of the common uses of cut is to improve efficiency of programs that have deterministic choices. For example, consider the following program that flips the case of atoms in the given list (`flip_case.pl`):

```
flip_case([], []).

flip_case([H|T], [H1|T1]) :-
        char_type(H, lower), upcase_atom(H, H1), flip_case(T, T1).

flip_case([H|T], [H1|T1]) :-
        char_type(H, upper), downcase_atom(H, H1), flip_case(T, T1).
```

Note: `char_type/2`, `upcase_atom/2` and `downcase_atom/2` are Prolog built-ins, use Prolog help to clarify the functionality.

The main point is that since a given character can be either upper case or lower case, only one between the second and the third clauses can ever be successfully applied at every stage of the search. Nevertheless, Prolog will try both of these clauses for every character in the list – this, potentially, could be very inefficient. To remedy this inefficiency, cuts can be placed in the following manner:

```
flip_case([], []).

flip_case([H|T], [H1|T1]) :-
        char_type(H, lower), !, upcase_atom(H, H1), flip_case(T, T1).

flip_case([H|T], [H1|T1]) :-
        char_type(H, upper), !, downcase_atom(H, H1), flip_case(T, T1).
```

Draw a search tree for the query `?- flip_case(['a','A','b'], L)` with and without cuts to better understand the difference.