Declarative problem solving methods

Lecture 4

<u>Overview</u>
Backtracking
Alternative answers
Cut !
Green and red cut
Negation \+
Forced backtracking fail

<u>References</u>
Bratko Chapter 5

Controlling backtracking

Backtracking
The Prolog interpreter starts a backtracking when
- it fails with an evaluation
- the user asks for alternative answers

Alternative answers
In the sport judo you compete in classes in relation to ones weight. Define a relation between a weight and a class.

Lightweight	up to 63 kilo
Welter-weight	64-70 kilo
Middle-weight	71-80 kilo
Light heavyweight	81-93 kilo
Heavyweight	more than 93 kilo

```
judo_class(W, lightweight) :-             (1)
  W =< 63.
judo_class(W, ’welter-weight’) :-         (2)
  W > 63,
  W =< 70.
judo_class(W, 'middle-weight') :-         (3)
  W > 70,
  W =< 80.
judo_class(W, 'light heavyweight') :-     (4)
  W > 80,
  W =< 93.
judo_class(W, heavyweight).               (5)

?- judo_class(65, Class).
Class =
```

Sometimes it is possible to decide that an alternative evaluation cannot succeed. One way to define when statement (5) should succeed is to include the premise about the weight.

```
judo_class(W, heavyweight) :-              (5')
     W > 93.

?- judo_class(65, Class).
Class =
```

```
p(X) :-
     q(X).                                  (6)
p(0).                                       (7)

q(2) :-
     r,
     s,
     t(2).                                  (8)
q(4) :-
     u(1).                                  (9)
q(6) :-
     x(1).                                 (10)

r.                                         (11)
s :-
     v(0).                                 (12)
s.                                         (13)

t(1).                                      (14)
u(1).                                      (15)
v(1).                                      (16)
x(1).                                      (17)

?- p(Y).
Y =
```

Cut

The search can be ruled in such a way that no alternative statement is tested if one of the clauses has succeeded. This can be done using *cut* which is denoted !. This construct has only meaning when backtracking.

Consider this simple program:

```
a :-
  b,
  c.

a.

b.

c.


?- a.
```

Now consider this:

```
a :-
  b,
  c.

a.

b.

c :-
  fail.    % fail always fails.


?- a.
```

And finally consider this:

```
a :-
  b,
  !,       % Disable backtracking.
  c.

a.

b.

c :-
  fail.    % fail always fails.


?- a.
```

```
judo_class(W, lightweight) :-               (1')
  W =< 63,
  !.
judo_class(W, ’welter-weight’) :-           (2')
  W > 63,
  W =< 70,
  !.
judo_class(W, middleweight) :-              (3')
  W > 70,
  W =< 80,
  !.
judo_class(W, 'light heavyweight') :-       (4')
  W > 80,
  W =< 93,
  !.
judo_class(W, heavyweight).                 (5')


?- judo_class(65, Class).

Class = welter-weight ;
```

Change sentence (8) by including ! after r.

```
p(X) :-
     q(X).                                  (6)
p(0).                                       (7)

q(2) :-
     r,
     !,
     s,
     t(2).                                 (8')
q(4) :-
     u(1).                                  (9)
q(6) :-
     x(1).                                 (10)

r.                                         (11)
s :-
     v(0).                                 (12)
s.                                         (13)

t(1).                                      (14)
u(1).                                      (15)
v(1).                                      (16)
x(1).                                      (17)
```

How is the search space changed?

```
?- p(Y).
Y =
```

Green and red cut

• If a cut is used and it does not influence the declarative interpretation you have a *green cut*.

• If a cut is used and it does influence the declarative interpretation you have a *red cut*.

Did we use a red or green cut for the program `judo_class`?

```
p :- a, b.
p :- c.
```

```
p :- c.
p :- a, b.
```

Logical intepretation:
p <- (a & b) V c
Even if the order of the statements changes they mean the same.

Example of a red cut:

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

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

Logical interpretation:
p <- (a & b) V (-a & c)
If we change the order the logical meaning will be
p <- c V (a & b)

Example of green cut:

```
min(T1, T2, T1) :-
  T1 < T2,
  !.
min(T1, T2, T2) :-
  T1 >= T2,
  !.
```

Negation

Example:
The predicate `intersection(List1, List2, List3)` is a relation between three lists where the third list, `List3`, should consist of only the elements that can be found in both the other two lists.

```
intersection([], _, []).                      (16)
intersection([E|R], List, [E|Set]) :-        (17)
  member(E, List),
  intersection(R, List, Set).
intersection([E|R], List, Set) :-            (18)
  intersection(R, List, Set).


member(X, [X|_]).                             (19)
member(X, [_|List]) :-                        (20)
  member(X, List).

?-  intersection(
      [1, 2, 3, 4], [2, 4, 6, 8, 10], L).
L =
```

All answers are not according to the definition. Cut can solve the problem.

```
intersection([], L, []).
intersection([E|R], List, [E|Set]) :-      (17')
  member(E, List),
  !,
  intersection(R, List, Set).
intersection([E|R], List, Set) :-          (18)
  intersection(R, List, Set).

?-  intersection([1, 2, 3, 4], [2, 4, 6, 8, 10], L).

L =
```

An alternative to cut is to describe in (18) that this clause is true if `E` is not a member in `List`.
\+ is built-in and means not, i.e. a negation.

Negation of a predicate means that if the predicate succeeds the evaluation fails. The negation is called *"negation as failure"* since the negation succeeds if Prolog is not able to prove that a statement is true and therefore the system concludes that it is false. The concept is not the same as negation in predicate calculus.

`\+ predicate`

`predicate` is proved to be true → fails

`predicate` cannot be proved to be true → succeeds

You can bind a variable when showing negation as failure.

```
p(1).

?- \+p(X).
no
```

The program intersection again…

```
intersection([], L, []).
intersection([E|R], List, [E|Set]) :-      (17)
     member(E, List),
     intersection(R, List, Set).
intersection([E|R], List, Set) :-          (18')
     \+member(E, List),
     intersection(R, List, Set).

?-  intersection([1, 2, 3, 4], [2, 4, 6, 8, 10], L).
L =
```

Forced backtracking

Let us define our own version of "negation as failure", `\+`. When a predicate P is fulfilled, then `negation(P)` fails. By using the built-in predicate `fail` then Prolog is forced to fail. The opposite predicate `true` which always succeeds.

```
negation(P) :-
  P, !, fail
  ;
  true.

dog('Karo',      5, labrador,  [black]).
dog('Pluto',    40, disney,    [yellow]).
dog('Lassie',    8, collie,    [brown,white,black]).
dog('Pongo',     4, dalmatian, [white,black]).
dog('Perdita',   3, dalmatian, [white,black]).
dog('Karolina', 4, labrador,  [yellow]).
```

```
?- negation(dog('Karo', 5, labrador, [black])).
```

```
?- negation(dog('Karo', 5, labrador, [brown])).
```

Alternative (but equivalent) version of negation:

```
negation(P) :-
  P, !, fail.
negation(_).
```

• Write a program that utilises `write` to list all the dogs.