G53KRR Handout on Prolog

Topics covered in the lecture were:

- Basics of Prolog
- Proof search/backtracking in Prolog
- Control of proof search
- Negation as failure

Prolog Prolog is a (logic) programming language where programs consist of facts and rules (Horn clauses):

```
parent(john, tom).
father(X, Y) :- parent(X,Y), male(X).
```

This is the same as:

$$Parent(john, tom)$$
$$\forall x \forall y (Parent(x, y) \wedge Male(x) \supset Father(x, y))$$

(Note that in Prolog variables are upper case and constants and predicate names are lower case. Every clause ends with a dot, and implications are pointing right to left.)

Proof search Consider the following knowledge base (from previous lecture, but in Prolog and with one extra rule for child/2 and one fact for baby/1 to please Swish):

```
toddler(jane).
female(jane).
baby(jim).

girl(X):- child(X), female(X).
child(X):-baby(X).
child(X):-toddler(X).
```

Suppose the query is `?- girl(X).`

Prolog reads the knowledge base, and tries to unify `girl(X)` (or rather something like `girl(_1111)` where `_1111` a new internally generated variable) with either a fact, or the head of a rule. It searches the knowledge base in the order the program is written, from top to bottom, and applies the first unification it finds, which is the head of `girl(X):- child(X), female(X).` This produces a new list of goals:

```
          ?- girl(X)
                |
    X=_1111     |
                |
         ?- child(_1111), female(_1111)
```

A list of goals is precessed left to right, so the next one to attempt to unify is `child(_1111)`. The first rule in order is the one with the baby, so we get:

```
           ?- girl(X)
               |
     X=_1111   |
               |
          ?- child(_1111), female(_1111)
              /
             /
            /
        ?- baby(_1111), female(_1111)
 _1111=jim |
        ?- female(jim)
           |
          fail
```

There is no way to prove `female(jim)`, so on this branch the search fails. Now Prolog *backtracks*: it goes back up to the nearest choice point to see if the was another choice for unifying `baby(_1111)` (no), then `child(_1111)` with a fact or a head of the rule. The toddler rule is the next one in the program, so there is indeed another choice, and for resulting list of goals we can unify against the facts in the knowledge base. Eventually we get this tree:

```
           ?- girl(X)
               |
     X=_1111   |
               |
          ?- child(_1111), female(_1111)
              /                     \
             /                       \
            /                         \
        ?- baby(_1111), female(_1111)  ?- toddler(_1111), female(_1111)
_1111=jim |                                      |  _1111 = jane
        ?- female(jim)                    ?- female(jane)
           |                                     |
          fail                                   ?-   (empty goal list=success)
```

Prolog will return `yes X = jane`. If we force Prolog to backtrack to find other values for X (hit Next in Swish), it will backtrack first to try and find another way of solving `toddler(_1111)` and then to another way of solving `girl(X)` but both will fail for the current knowledge base.

You can see the Prolog trace in Swish if you use Solutions -> Debug (trace).

Controlling search .

How much Prolog has to backtrack and in general how much effort the search takes depends on the order of clauses in the program. For example, given the following knowledge base:

```
parent(a,b).
parent(b,c).
parent(c,d).
ancestor(X,Y):-parent(X,Y).
```

the following three choices for the second rule in the definition of ancestor are logically equivalent:

```
ancestor(X,Y):-ancestor(X,Z),ancestor(Z,Y).

ancestor(X,Y):-ancestor(X,Z),parent(Z,Y).

ancestor(X,Y):-parent(X,Z), ancestor(Z,Y).
```

However the first two will cause an infinite recursion/stack overflow if forced to backtrack after the last solution is found. The last one works best and simply returns false if forced to backtrack after the last solution `X=b,Y=d` is found.

Another example of the importance of clause ordering is given in Brachman and Levesque Chapter 6. If we ask a query `am_cousin(X,sally)` from a knowledge base where there are lots of Americans and only a handful of cousins of Sally, it is much better to use

```
am_cousin(X,Y): - cousin(X,Y), american(X).
```

than

```
am_cousin(X,Y): - american(X), cousin(X,Y).
```

because the former will backtrack only over cousins of Sally, and the latter over most of Americans.

Another way to control search is to explicitly control backtracking using cut: !. Cut is a goal that always succeeds and has a side effect that backtracking is prevented above the cut in the search tree: Prolog commits to any choices made until the cut was encountered. So,

```
G:- T1, ..., Tm, !, S1,...,Sn
```

means that Prolog should not backtrack after finding unifications for T1, ..., Tm (if it fails somewhere with S1,...,Sn, it does not make sense to look for other variable bindings and try to unify T1,...., Tm with heads of rules lower in the program).

I gave over-complicated examples in lecture 10; this one from Learn Prolog Now! is better (it stops the program trying to find the max of X and Y in another way after it succeeded once:

```
max(X,Y,Y)  :-  X  =<  Y,!.
max(X,Y,X)  :-  X>Y.
```

Negation as failure In Prolog, we can use negation (in Swish it is `not`) but it is different from logical negation. `not(G)` succeeds if `G` fails (we cannot prove `G` from the knowledge base). Clearly in classical logic if α does not follow from KB, this does not mean that $\neg\alpha$ follows: maybe neither follows. But negation as failure is a useful way of reasoning. If we look at a bus timetable and see that there are no buses after 18:00 in the timetable, we conclude that there are no buses after 18:00 in reality. We will study semantics of this kind of reasoning later in the module.

Negation in Prolog can be defined as

```
not(G)  :-  G,!,fail.
not(G).
```

First we try to prove G, and if this succeeds, then `not(G)` fails. If the first clause does not succeed (we cannot prove G) then the second clause is used, and `not(G)` succeeds.

In lecture 10, I defined `nochildren/1` as

```
nochildren(X):- not(parent(X,Y)).
```

This does not quite work in Swish syntactically (it complains about `Y` only occurring once) and does not work for queries with variables at all. What works is the following program (note the use of `_` for variables that only occur once, and making sure that X is instantiated before we get to negation – otherwise the goal is parent(X,Y) and it succeeds because there are suitable values for X and Y).

```
parent(a,b).
parent(b,c).
parent(c,d).
person(a).
person(b).
person(c).
person(d).

nochildren(X):- person(X), not(parent(X,_)).
```