Tutorial SHEET-2

Matching and Proof Search

1. Matching

We define when two terms match in Prolog. The basic idea is:
Two terms match, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal.

That means that the terms mia and mia match, because they are the same atom. Similarly, the terms 42 and 42 match, because they are the same number, the terms X and X match, because they are the same variable, and the terms woman(mia) and woman(mia) match, because they are the same complex term. The terms woman(mia) and woman(vincent), however, do not match, as they are not the same (and neither of them contains a variable that could be instantiated to make them the same).
For the terms mia and X, even though they are not the same, the variable X can be instantiated to mia which makes them equal. So, by the second part of the above definition, mia and X match.
Similarly, the terms woman(X) and woman(mia) match.

However, for loves(vincent,X) and loves(X,mia), it is impossible to find an instantiation of X that makes the two constant terms equal, and therefore they don't match.

When Prolog matches two terms it performs all the necessary instantiations, so that the terms really are equal afterwards.

Thus, here's a more precise definition for matching:

1. 1. If term1 and term2 are constants, then term1 and term2 match if and only if they are the same atom, or the same number.
2. If term1 is a variable and term2 is any type of term, then term1 and term2 match, and term1 is instantiated to term2. Similarly, if term2 is a variable and term1 is any type of term, then term1 and term2 match, and term2 is

instantiated to term1. (So if they are both variables, they're both instantiated to each other, and we say that they share values.)

3. If term1 and term2 are complex terms, then they match if and only if:
    1) They have the same functor and arity.
    2) All their corresponding arguments match
    3) and the variable instantiations are compatible. (i.e. it is not possible to instantiate variable X to mia, when matching one pair of arguments, and to then instantiate X to vincent, when matching another pair of arguments.)
4. Two terms match if and only if it follows from the previous three clauses that they match.

1.1. Examples

We can check for matching in Prolog using the built-in predicate, =/2 predicate (recall that the /2 at the end is to indicate that this predicate takes two arguments). The =/2 predicate tests whether its two arguments match.

For example, if we pose the query
=(mia,mia).
Prolog will respond 'yes', and if we pose the query
=(mia,vincent).
Prolog will respond 'no'.
The predicate can also be used in **infix** notation. Suppose the query posed is:
mia = mia.
yes

Consider the following query:
'mia' = mia.
yes
As far as Prolog is concerned, 'mia' and mia are the same atom. In fact, for Prolog, any atom of the form 'symbols' is considered the same entity as the atom of the form symbols.

On the other hand, to the query:
'2' = 2.

Prolog will respond ‘no’. This is because 2 is a number, but ’2’ is an atom. They simply cannot be the same.
Let’s try an example with a variable:
mia = X.
X = mia
yes
Here, the variable is instantiated to the first argument, namely mia.

Consider the query:
X = Y.
yes
Prolog simply agrees that the two terms unify and makes a note that from now on, X and Y denote the same object. That is, if ever X is instantiated, Y will be instantiated too, and to the same thing. The output could also be:
X = _5071
Y = _5071
where _5071 is a variable. Variables in Prolog are denoted by _(number). Thus here Prolog has assigned a common variable for both the variable terms.

X = mia, X = vincent.
Prolog will respond ’no’. This query involves two goals, X = mia and X = vincent.

Look at an example involving complex terms:
kill(shoot(gun),Y) = kill(X,stab(knife)).
X = shoot(gun)
Y = stab(knife)
yes
Clearly the two complex terms match if the stated variable instantiations are carried out.

Also try the query:
loves(X,X) = loves(marcellus,mia).

1.2. The occurs check

Consider the following query:
father(X) = X.

These terms do not match according to the standard unification algorithm. With old Prolog implementations you would get a message like:

Not enough memory to complete query!

and a long string of symbols like:

X = father(father(father(father(father(father(father(father (father(fathe

Prolog repeatedly tries to match these terms, but it won't succeed.

Current Prolog implementations have found a way of coping with this problem and give the answer:

X = father(father(father(father(father(father(...))))))))))

Standard unification algorithms use **occurs check** by peeking inside the structure of the terms they are asked to unify, looking for strange variables (like the X in our example) that would cause problems.

Prolog, on the other hand, assumes that you are not going to give it anything dangerous. So it does not make an occurs check. As soon as you give it two terms, it charges full steam ahead and tries to match them. This is because carrying out an occurs check every time matching was called for would slow it down considerably.

1.3 Programming with Matching

Let's add the following rules to a knowledge base:

```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).
```

Here, right down at the bottom level, we have a complex term with a functor point and two arguments. Its two arguments are intended to be instantiated to numbers: point(X,Y) represents the Cartesian coordinates of a point. That is, the X indicates the horizontal distance the point is from some fixed point, while the Y indicates the vertical distance from that same fixed point.

Therefore, trying the following queries, we get back the following outputs:

```
vertical(line(point(1,1),point(1,3))).
yes
```

```
vertical(line(point(1,1),point(3,2))).
no

horizontal(line(point(1,1),point(2,Y))).
Y = 1 ;
no

horizontal(line(point(2,3),P)).
P = point(_1972,3) ;
no
```

In the last query, Prolog correctly says that the y coordinate for a horizontal line is 3 and gives a variable to the x coordinate.The answer is a structured term and was built using matching and nothing else: no logical inferences were used to produce it.

2. Proof Search

Suppose we are working with the following knowledge base

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X),g(X),h(X).
```

Suppose we then pose the query

```
k(X).
```

Let's see how Prolog works out this query.

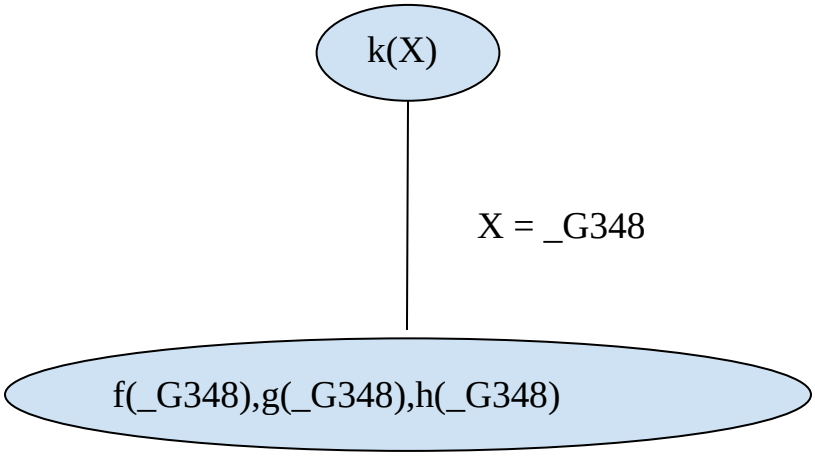
Prolog reads the knowledge base, and tries to match `k(X)` with either a fact, or the head of a rule. It searches the knowledge base top to bottom, and carries out the matching, if it can, at the first place possible. Here there is only one possibility: it must match `k(X)` to the head of the rule

```
k(X) :- f(X),g(X),h(X).
```

When Prolog matches the variable in a query to a variable in a fact or rule, it generates a brand new variable to represent that the variables are now sharing. So the original query now reads:

```
k(_G348)
```

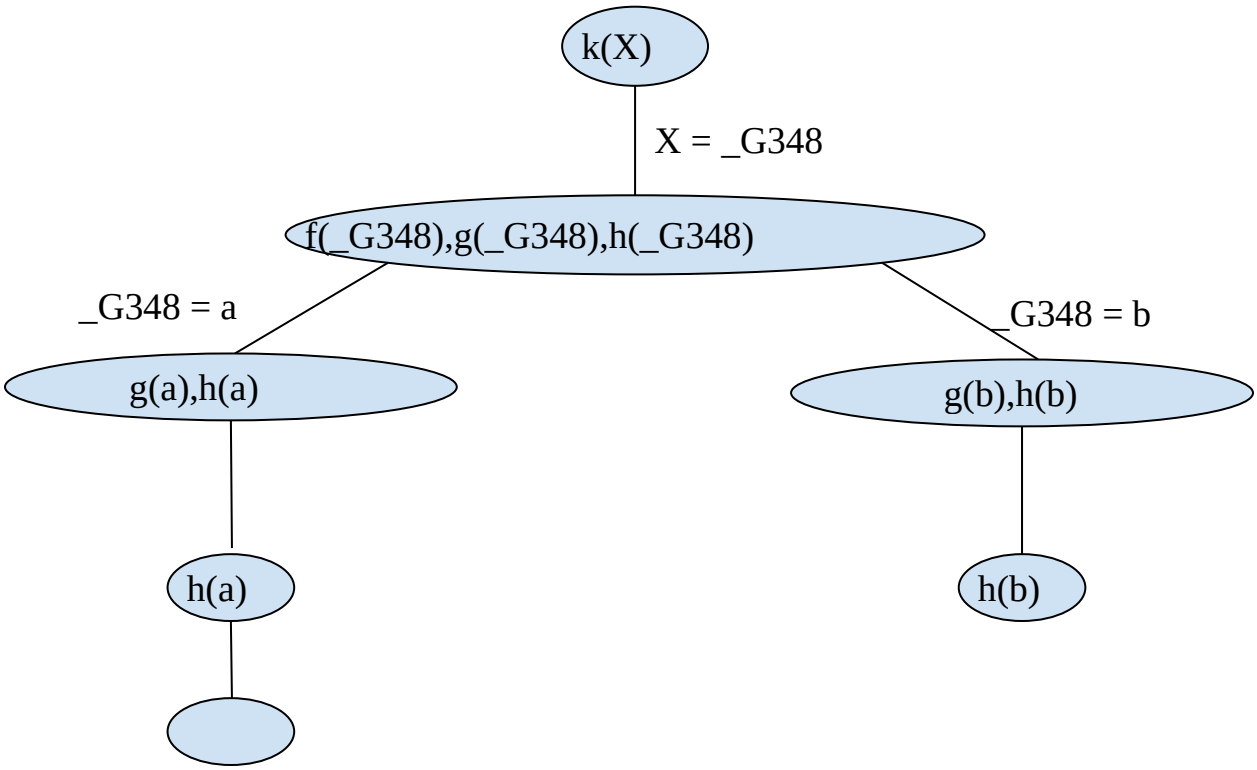
and Prolog knows that
k(_G348) :- f(_G348),g(_G348),h(_G348).
So if Prolog can find an individual with properties f, g, and h, it will have satisfied the original query. So Prolog replaces the original query
with the following list of goals:
f(_G348),g(_G348),h(_G348)
We will represent this graphically as



Now, whenever it has a list of goals, Prolog tries to satisfy them one by one, working through the list in a left to right direction. The leftmost goal is f(_G348)
It tries to satisfy it by searching through the knowledge base from top to bottom. The first thing it finds that matches this goal is the fact f(a). This satisfies the goal f(_G348) and we are left with two more goals to go. When matching f(_G348) to f(a), X is instantiated to a. This applies to all occurrences of X in the list of goals. So, the list of remaining goals is:
g(a),h(a)
The fact g(a) is in the knowledge base. So the next goal we have to prove is satisfied too, and the goal list is now
h(a)
But there is no way to satisfy this goal. The only information h we have in the knowledge base is h(b) and this won't match h(a).
So Prolog decides it has made a mistake and checks whether at some point there was another possibility for matching a goal with a fact or the head of a rule in the knowledge base. It does this by going back up the path in the graphical representation that it was coming down on. There is nothing else in the knowledge base that matches with g(a), but there is another possibility for matching f(_G348).

Points in the search where there are several alternatives for matching a goal against the knowledge base are called **choice points**. Prolog keeps track of choice points and the choices that it has made there, so that if it makes a wrong choice, it can go back to the choice point and try something else. This is called **backtracking**.
So, Prolog backtracks to the last choice point, where the list of goals was:
f(_G348),g(_G348),h(_G348).
Prolog has to redo all this. Prolog tries to **resatisfy** the first goal, by searching further in the knowledge base. It sees that it can match the first goal with information in the knowledge base by matching
f(_G348) with f(b). This satisfies the goal f(_G348) and instantiates X to b, so that the remaining goal list is
g(b),h(b).
But g(b) is a fact in the knowledge base, so this is satisfied too, leaving the goal list:
h(b).

And this fact too is in the knowledge base, so this goal is also satisfied. Prolog now has an empty list of goals. This means that it has proved everything it had to to establish the original goal, namely k(X). So this query is satisfiable, and moreover, Prolog has also discovered what it has to do to satisfy it, namely instantiate X to b.
Representing these last steps graphically gives us



Proof search can also be done using the built-in Prolog predicate trace. This is an built-in Prolog predicate that changes the way Prolog runs: it forces Prolog to evaluate queries one step at a time, indicating what it is doing at each step. Prolog waits for you to press return before it moves to the next step, so that you can see exactly what is going on. It was really designed to be used as a debugging tool, but it's also really helpful when you're learning Prolog: stepping through programs using trace is an excellent way of learning how Prolog proof search works.

Let's look at an example. We looked at the proof search involved when we made the query k(X) to the following knowledge base:

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X),g(X),h(X).
```

Suppose this knowledge base is in a file proof.pl. We first consult it:

```
1 ?- [proof].
% proof compiled 0.00 sec, 1,524 bytes
```

yes
We then type ‘trace.’ and hit return:
2 ?- trace.
Yes

Prolog is now in trace mode, and will evaluate all queries step by step. For example, if we pose the query k(X), and then hit return every time Prolog comes back with a ?, we obtain (something like) the following:

```
[trace] 2 ?- k(X).
Call: (6) k(_G348) ?
Call: (7) f(_G348) ?
Exit: (7) f(a) ?
Call: (7) g(a) ?
Exit: (7) g(a) ?
Call: (7) h(a) ?
Fail: (7) h(a) ?
Fail: (7) g(a) ?
Redo: (7) f(_G348) ?
Exit: (7) f(b) ?
Call: (7) g(b) ?
Exit: (7) g(b) ?
Call: (7) h(b) ?
Exit: (7) h(b) ?
Exit: (6) k(b) ?
X = b
Yes
```

Study this carefully. Try to relate this output to the discussion of the example above.
To turn the trace off, simply type ‘notrace.’ and hit return:
notrace.

3. Exercises

1. Which of the following pairs of terms match? Where relevant, give the variable instantiations that lead to successful matching.
    1) bread = bread
    2) 'Bread' = bread
    3) 'bread' = bread
    4) Bread = bread
    5) bread = sausage
    6) food(bread) = bread
    7) food(bread) = X
    8) food(X) = food(bread)
    9) food(bread,X) = food(Y,sausage)
    10) food(bread,X,beer) = food(Y,sausage,X)
    11) food(bread,X,beer) = food(Y,kahuna_burger)
    12) food(X) = X
    13) meal(food(bread),drink(beer)) = meal(X,Y)
    14) meal(food(bread),X) = meal(X,drink(beer))

2. Make sure you understand the way \=/2 predicate works by trying it out on (at least) the following examples:
    1) a \= a
    2) 'a' \= a
    3) A \= a
    4) f(a) \= a
    5) f(a) \= A
    6) f(A) \= f(a)
    7) g(a,B,c) \= g(A,b,C)
    8) g(a,b,c) \= g(A,C)
    9) f(X) \= X

Thus the \=/2 predicate is (essentially) the negation of the =/2 predicate.

3. Here is a tiny lexicon and mini grammar with only one rule which defines a sentence as consisting of five words: an article, a noun, a verb, and again an article and a noun.

```
word(article,a).
word(article,every).
word(noun,criminal).
word(noun,'big kahuna burger').
```

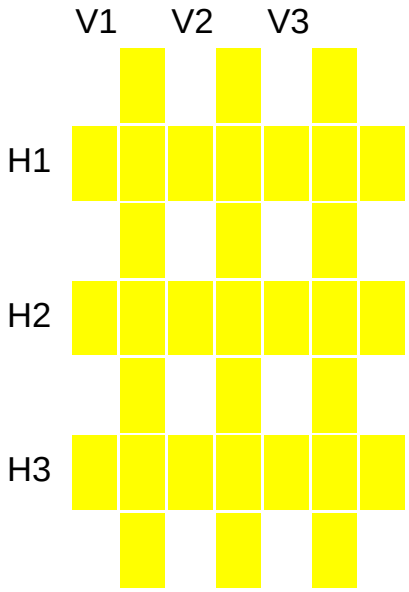
```
word(verb,eats).
word(verb,likes).
sentence(Word1,Word2,Word3,Word4,Word5) :-
word(article,Word1),
word(noun,Word2),
word(verb,Word3),
word(article,Word4),
word(noun,Word5).
```

What query do you have to pose in order to find out which sentences the grammar can generate? List all sentences that this grammar can generate in the order Prolog will generate them. Make sure that you understand why Prolog generates them in this order.

4. Here are six English words:

abalone, abandon, anagram, connect, elegant, enhance.

They are to be arranged in a crossword puzzle like fashion in the grid given below.



The following knowledge base represents a lexicon containing these words.

```
word(abalone,a,b,a,l,o,n,e).
word(abandon,a,b,a,n,d,o,n).
word(enhance,e,n,h,a,n,c,e).
```

word(anagram,a,n,a,g,r,a,m).
word(connect,c,o,n,n,e,c,t).
word(elegant,e,l,e,g,a,n,t).

Write a predicate crosswd/6 that tells us how to fill the grid, i.e. the first three arguments should be the vertical words from left to right and the following three arguments the horizontal words from top to bottom.