Prolog Lab: Matching and Unification. Trace.

Much of the content of this tutorial is adapted from http://www.learnprolognow.org/

Prolog Matching: the built-in predicates = and \=

Prolog provides a built-in predicate which takes two arguments and check whether they match. This predicate is =/2. How does Prolog answer the following queries? Think about it first, and then type them into Prolog to check whether you are right.

```
?- =(harry,harry).

?- =(harry,'Harry').

?- =(f(a,b),f(a(b))).
```

Prolog also allows you to use = as an infix operator. So, instead of writing =(harry,harry) you can write harry = harry.

Matching in Prolog is a destructive operation because variables get instantiated. So, the terms that are matched may be different after matching to what they were before matching. Here is a sequence of queries that illustrates this.

```
?- X = harry, Y = hermione.

X = harry

Y = hermione;

no
```

Matching the variable X with harry and the variable Y (a different variable) with hermione works and instantiates both variables.

```
?- X = harry, X = hermione.

no
```

After the first goal X = harry has been processed, the variable is instantiated to harry. harry does not match with hermione (different atoms), so that the second goal fails.

The built-in predicate \=/2 works in the opposite way: *Arg1* \= *Arg2* is true if *Arg1* = *Arg2* fails, and vice versa. (Note that not all Prolog implementations provide \=/2. SWI Prolog does, but Sicstus, for example, doesn't.)

Try some queries involving = and \=.

Unification in Prolog

We've so far seen many examples when we run queries in Prolog using a variable, and that variable is instantiated. For example Prolog **unifies** woman(X) with woman(mia) thereby instantiating the variable **X** with the atom **mia**.

Keep in mind that Prolog **does not evaluate** terms (unless it's explicitly requested to do so).

```
?- x=10+5.

false.

?-
```

Prolog carries out a left to right depth first search of a database and tries to **match** (or unify) terms based on the syntax of those terms. When Prolog unifies two terms it performs all the necessary **instantiations** (substitutions/bindings), so that the terms are equal afterwards.

```
?- like(X,Y) = like(students, prolog).
X = students,
Y = prolog.
?-
```

When do two terms unify?

1. *If T1 and T2 are constants, then T1 and T2 unify if they are the same atom, or the same number.*

   Two atoms match if they are the same atom, two numbers match is they are the same number.
   - harry = harry
   - harry != 'Harry'

2. *If T1 is a variable and T2 is any type of term, then T1 and T2 unify, and T1 is instantiated to T2. (and vice versa)*

   A variable matches any other term. The variable gets instantiated with the other term.
   - X=wizard(harry)
   - X=Y

3. *If T1 and T2 are complex terms then they unify if:*
   a) They have the same functor and arity, and
   b) all their corresponding arguments unify, and
   c) the variable instantiations are compatible.

   - like(harry, hargrid) = like(harry, X)
   - like(harry, hargrid) != like(harry, X, Y)
   - like(harry, hargrid) != like(X, X)

What about the terms loves(mia, X) and loves(X,vincent)?

```
?- loves(mia, X) = loves(X, vincent).
false.
```

Answer
X can't be instantiated with both mia and vincent. A variable can only take on one binding within a query (not including backtracking conditions). So, after working through the first goal, Prolog has instantiated X with **mia**, so that it cannot unify it

Few more examples with complex terms – try them out:

```
?- k(s(g), Y) = k(X, t(k)).
X=s(g)
Y=t(k)
yes


?- k(s(g), t(k)) = k(X, t(Y)).
X=s(g)
Y=k
yes
?-
```

Finally, consider the following query:

```
?-father(X) = X.
```

Does these terms unify or not? Try it out.

```
?- father(X) = X.
X=father(father(father(father(father(father(father(father(father(father(father(fat
her(father(father(father(father(father(father(father(father(father(father(father(fa
ther(father(father(father(father(father(father(…
```

Prolog does not use the standard unification algorithm – a standard algorithm will return that such turns don't unify – it will instantiate the variable X with, let's say, bill, and will evaluate that father(bill) and bill don't unify. It will realise that the left-handside will always be longer than the right-handside.
The Prolog unification algorithm won't spot that. Following our rules for unification above, X will be instantiated to father(X). However, there is an X here that has already been instantiated to father(X), so X will be instantiated to father(father(X)), etc.
Finally, Prolog provides a predicate that will carry out standard unification, that is unification with occurs that checks for recursive use of the variable in the unification.

```
?-  unify_with_occurs_check(father(X),X).
false.
```

Exercise: Which of the following pairs of terms unify? Where relevant, give the variable instantiations that lead to successful unification.

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, burger)
12. food(X) = X
13. meal(food(bread),drink(beer)) = meal(X,Y)
14. meal(food(bread),X) = meal(X, drink(beer))

Trace: watching how Prolog works

trace allows the user to monitor the progress of the Prolog interpreter. This monitoring is accomplished by printing to the output every goal that Prolog attempts. The tracing facilities in Prolog are often rather cryptic and take some study and experience to understand.

The information available in a trace of a Prolog program usually includes the following:

1. The depth level of recursive calls (marked left to right on line).
2. When a goal is tried for the first time (this is indicated by the term Call)
3. When a goal is successfully satisfied (this is indicated by the term Exit)
4. When a goal has further matches possible (indicted by the term Redo or in some Prolog implementations retry).
5. When a goal fails because all attempts to satisfy it have failed.
6. The goal notrace stops the exhaustive tracing.

Load the knowledge base about Harry Potter from the previous labs. Then type trace, and hit return:

```
?- trace.

true.
```

Now, Prolog is in trace mode and will show you step by step how it is searching for answers. For example, pose the query happy(aunt_petunia). Prolog will show you the first goal that is sets out to prove:

```
Call: (7) happy(aunt_petunia)
```

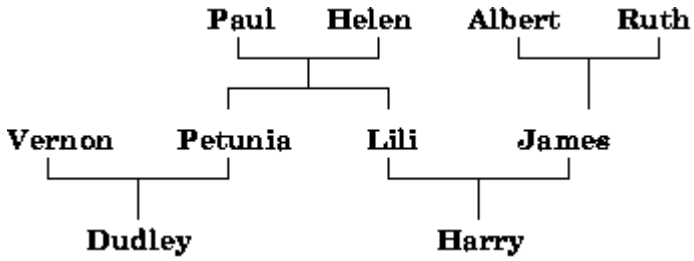
Hitting return will show you the next step. In lines starting with CallProlog is telling which is the goal that it is at the moment attempting to prove. Fail: ... means that the specified goal failed, and Exit: ... means that it succeeded. With Redo: ... Prolog is

telling that it is trying to find an alternative way of proving a goal which it has already attempted to prove before.

Finish stepping through this trace and do traces for the other queries; for example, pose the query happy(X).

notrace turns the trace mode off. In SWI Prolog, the trace mode is only active for one query after typing trace.

Exercise:
Use the predicates male/1, female/1, and parent_of/2 to represent the following family tree as a Prolog knowledge base.



Now, formulate rules to capture the following relationships:

- father_of(Father,Child) and mother_of(Mother,Child)
- grandfather_of(Grandfather,Child) and grandmother_of(Grandmother,Child)
- sister_of(Sister,Person)
- aunt_of(Aunt,Person) and uncle_of(Uncle,Person)

Test your knowledgebase with questions like these:

1. *Does Harry have an aunt? Who?*
2. *Who are the grandparents of Harry?*
3. *Who are the grandchildren of Paul and Helen?*
4. *Does James have a sister?*

If you have finished early, extend the family tree above, for example add a sister for James, and test the questions again.