Division of Computing Science and Mathematics

CSCU9Y4 Prolog Practical 2

More complex goals
2. Have a look at the predicates `parent` and `grandfather`.

```
parent(X,Y) :- father(X,Y); mother(X,Y).
grandfather(X,Y) :- father(X,Z), mother(Z,Y).
grandfather(X,Y) :- father(X,Z), father(Z,Y).
```

Notice that the `grandfather` rule has two rules. If Prolog cannot satisfy a goal using the first rule of `grandfather` it will attempt to use the second rule. Try out some goals involving the predicates `'parent'` and `'grandfather'`. For example

```
?- parent(dan,phil).
?- parent(phil,dan).
?- parent(jenny,X).
?- parent(X,elizabeth).
?- grandfather(phil,josh).
?- grandfather(phil,X).
?- grandfather(X,tony).
```

Try tracing some of the goals to see which goals the interpreter has to achieve on the way to achieving the requested goal (use `trace.`, and turn it off with `notrace. nodebug.`). This will help you gain a better understanding of how Prolog works.

3. Have a go at writing a predicate `granddaughter(V,W),` which determines whether V is the granddaughter of W and writes a suitable message, e.g. `beatrice is the granddaughter of elizabeth`. Add this new predicate to your `archers` file and `consult` the file. Try some suitable goals:

```
?- granddaughter(shula,doris).
?- granddaughter(debbie,jack).
?- granddaughter(X,peggy).
?- granddaughter(kenton,doris).
```

The answers should be Yes, Yes, helen and No.

Use of the semicolon during goal execution
4. Last time you entered the goal

```
?- mother(jill,X).
```

Try it again, but this time key a semicolon after each response from the interpreter (note that it gives you the opportunity to do this only in cases where the goal contained a variable). You should see all the children of `'jill'` appearing. Using the semicolon forces Prolog to try to re-achieve the goal. This continues until the goal can no longer be achieved. It gives a form of repetition, but not a very useful one, since it relies on the user typing ";".
[Note: using a semicolon as part of a rule has a different meaning: OR]

Failure Driven Loops
5. Now try the following. See if you can explain what happens.

```
?- father(dan,X), write(X), nl, fail.
```

The goal 'fail' does exactly what its name says: it always fails, thereby causing Prolog to backtrack. This means that Prolog will try and reachieve the goal. (So it's a bit like an automated way of typing ; at the prompt).

Notice that when all the children of dan are written out, Prolog says "no". This is because it fails the last goal it tries. The following example shows how to make this failure driven loop terminate in a more elegant fashion. It's also more generic since it deals with the children of parents other than dan.

6. Edit the 'archers.plg' file, to add the following:

```
write_children(X) :-   parent(X,Y),
                       write(Y), nl,
                       fail.
write_children(X) :-   write('End of list'), nl.
```

Do another 'consult' as before, and then try out some goals involving 'write_children'.

7. Write a predicate write_siblings(X) using a failure driven loop to write out all the siblings of a person (male or female). Don't include the person themselves! Make sure your predicate terminates nicely. Test your predicate with the following goals:

```
?- write_siblings(josh).
?- write_siblings(shula).
?- write_siblings(daniel).
```

The answers should be
pip, End of list.
kenton, david, elizabeth, End of list.
End of list.
You will get duplicates in your lists. This is acceptable, but you need to explain why you are getting duplicates.

Recursion!
8. Issue the command

```
?- listing(ancestor).
```

and examine the ancestor rule. Note that ancestor appears both in the head and the body of the rule (i.e. the predicate calls itself), but that the arguments are different (otherwise how will it terminate?).

Now try some goals involving the predicate 'ancestor':

```
?- ancestor(doris,david).
```

```
?- ancestor(phil,jenny).
?- ancestor(tony,peggy).
```

Think about what is going on here (look at the first goal again, using `trace` to help. Observe that some goals have to be redone because of backtracking).

9. How might you use the ancestor predicate to
   a) find an ancestor of pip?
   b) find a descendant of phil who is at least two generations away (i.e. not a son or daughter)?
   c) display all descendants of jack?

Write goals for these queries. Convince yourself you have the right answers by inspecting the database facts.

Findall

10. `'findall'` is a very useful predicate which is built in. See if you can get a feel for what `'findall'` does by experimenting with it in conjunction with the `'archers'` database. So consult the `'archers'` file, and try these goals:

```
?- findall(X,grandfather(phil,X),L).
?- findall(X,(ancestor(dan,X), female(X)),L).
?- findall([X,Y],mother(X,Y),L).
```

Compare the first one with the goal `grandfather(phil,X)` and using ";" to return more answers. What do you get in `L`? The second goal shows that we can use a compound predicate to filter the list, and in the third one the list generated is a list of `[mother,child]` pairs. We can use `'findall'` to accumulate a list of terms of any kind, not just simple values.

11. Construct the following goals involving `'findall'`
   a) generate a list of all ancestors of pip
   b) generate a list of all descendants of pip

Checkpoint

Now demonstrate to a tutor that you have successfully created the granddaughter predicate, write_siblings predicate, queries for the ancestor predicate, and queries with findall. (Steps 3, 7, 9, 11 above)

Don't stop now, there's more overleaf.

Setof
Another useful built-in function is `'setof'`. Like `'findall'` it has three arguments: `setof(X,predicate(X),L)`, where `L` is a list of `X` items all satisfying `predicate(X)`. Unlike `'findall'`, duplicates are automatically removed. Use `'setof'` to create a version of `write_siblings` which does not produce duplicates.

More recursion!
Consult the file called `'mow.plg'`. To see what the program does, give the goal

```
?- poem(4).
```

Now look at the contents of the file (via WordPad, as before), and try to understand what it does. As suggested in a comment in the file, try to work out on paper the sequence of subgoals that the interpreter will work through in response to the goal `?- poem(1).` Note that `'poem'` is the "top-level" predicate, and the others (`'verses'`, `'writeVerse'`, `'writeWord'`, `'writeRest'`, `'writeBody'` and `'writeLast'`) are subsidiary predicates. Note further that these represent subsidiary tasks (compare with the use of methods in Java). You can also use `trace` to see the sub-goals, but it's a bit confusing because the output is mixed in with the trace.

Note that `'verses'` is recursive. How does it terminate?

Lists
Try out the recursive predicates `'sumlist'` and `'duplicate'`, as given in the lectures. They are in the file `'recursion.plg'`. Consult `recursion` and have a look at them first using `listing(sumlist)` and `listing(duplicate).`

Issue the following goals:

```
?- sumlist([6,2,3,4],S).
?- sumlist([5],S).
?- sumlist([0,0,0,0,0],S).
?- sumlist([],S).
```

Convince yourself you know what is going on. Remember the 'trace' facility. Try using this to see what steps Prolog must go through to achieve your goals. Do the same thing for goals involving `duplicate`.

Construct a predicate `count_generation(X,N)` to count the number of generations `N` above an individual `X`. You will need the `father` predicate, and a recursive call to `count_generation`.

Extra Practice
If you're feeling adventurous, and want to test your understanding of Prolog there are other Prolog files containing facts and rules for you to try. Examine:

ages.pl

greenbottles.pl (this file has some deliberate errors, to exercise your Prolog skills. There are 5 syntax errors, and 2 errors in the control of the program. The former will be picked up by the compiler, so look at the error messages carefully when you consult the file. The latter will be apparent when you run the program and it fails to produce the expected output.)