

## CS2910: INTRODUCTION TO PROLOG

### Exercise 2: ‘Databases’ and ‘Recursion’\*

Weeks 2 and 3

#### I. Prolog Database

This exercise is based on a database representing the fragment of a family tree illustrated in Figure 1.

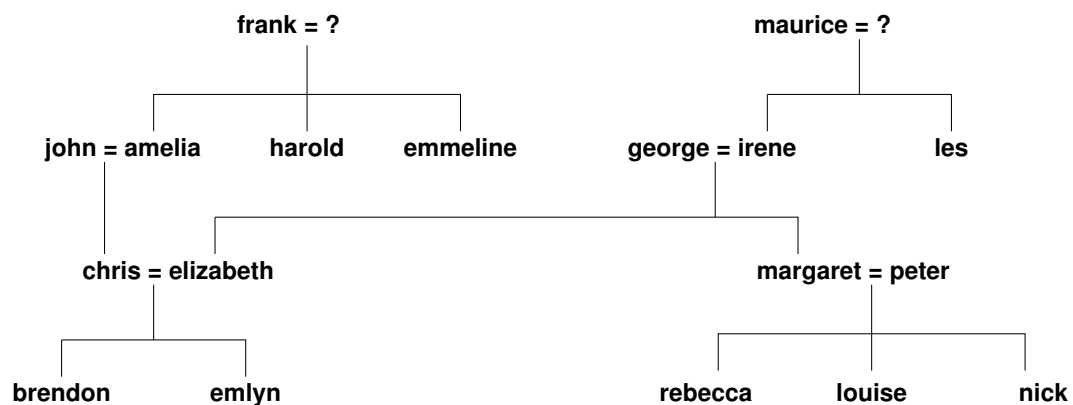


Figure 1: Family Tree

The database is the set of atomic facts contained in the file `family.pl`. Download a copy from Moodle. Load it in SWI-Prolog with `consult(family).` or `[family].`. The file contains clauses about three predicates:

<code>child_of( X, Y )</code>	"X is a child of Y"
<code>male( X )</code>	"X is male"
<code>female( X )</code>	"X is female"

For instance, the first fact in the database is

```
child_of( emmeline, frank ).
```

and expresses “emmeline is a child of frank”.

---

\*Thanks to **Kostas Stathis**, Marek Sergot and Chris Hogger.

1. Enter the following queries, finding all the solutions:

```
child_of( peter, irene ).
child_of( peter, emlyn ).
child_of( X, george ).
child_of( george, Y ).
child_of( X, X ).
child_of( X, Y ).
```

2. Add further Prolog clauses which define the following predicates:

<code>mother_of( M, X )</code>	"M is the mother of X"
<code>grandparent_of( GP, X )</code>	"GP is a grandparent of X"
<code>daughter_of( D, X )</code>	"D is a daughter of X"
<code>uncle_of( Unc, X )</code>	"Unc is an uncle of X"
<code>niece_of( N, X )</code>	"N is a niece of X"
<code>great_grandfather_of( Gfx, X )</code>	"Gfx is a great-grandfather of X"
<code>ancestor_of( Anc, X )</code>	"Anc is an ancestor of X"

For the last one, compare the clauses for `superiorOf/2` in the lecture notes.

Test each set of clauses by posing suitable queries and finding all their solutions. As a guiding example, this is how we might define "X is a sister of Y":

```
sister_of( X, Y ) :-
    child_of( X, Z ),
    child_of( Y, Z ),
    X \= Y,
    female( X ).
```

(This definition is for illustration: it treats step-sisters as sisters.)

Note that in Prolog `\=` means 'do not unify'. You could also use `\==` (backslash, equals, equals) which means 'not identical'. However, you are advised not to. Although you see `\==` in many books it is an *extra-logical* primitive and it must be used with care.

(To appreciate the difference, compare what you get when executing Prolog queries

```
?- a \= b.
?- a \== b.
?- X \= a.
?- X \== a.
?- X \= Y.
?- X \== Y.
```

And compare:

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

So from a logic/declarative perspective `==` and `\==` are horrible things.)

## II: Recursion

**Notation** When specifying a relation/program to be defined by Prolog clauses in this and subsequent exercises we will use argument annotations to indicate the intended uses of the definition. We write  $+X$  to indicate that an argument  $X$  will be given,  $-X$  to indicate that the argument  $X$  will be an unbound variable, and  $?X$  to indicate that it can be given or be an unbound variable. This is the notation used in the SWI-Prolog manual when describing uses of the primitive ('built in') predicates.

1. An acyclic directed graph is represented by the following facts:

```
arc(a, b).      arc(b, c).      arc(b, d).
arc(c, f).      arc(d, f).      arc(c, e).
arc(f, e).
```

You can find these in a file `graph.pl` via Moodle or type them in yourself.

- (a) Write two Prolog clauses in order to define the predicate `path( ?X, ?Z )` which holds when the graph contains a uniformly-directed path from `X` to `Z`. For instance, in the given graph there is a path from `b` to `f`.  
Your program should refer only to the `path` and `arc` predicates.
  - (b) Try out your program for the queries:
    - i. `path(b, f).`
    - ii. `path(b, Z).`
    - iii. `path(X, d).`
  - (c) Enter and run queries asking if
    - i. there is a path from `a` to `f` that passes through `d`;
    - ii. there is a path of length 3 ending at `f` (assume each arc has unit length);
    - iii. there is a cycle in the graph.
2. The Peano numbers are `0`, `s(0)`, `s(s(0))`, ... etc. representing 0, 1, 2, ... etc.
    - (a) Write a Prolog program for `plus( ?X, ?Y, ?Z )` which holds when `X`, `Y` and `Z` are Peano numbers satisfying  $X + Y = Z$ . Use no predicate other than `plus`.
    - (b) Try out your program for the queries
      - i. `plus(s(0), s(s(0)), Z).`
      - ii. `plus(s(0), Y, s(s(s(s(0))))).`
      - iii. `plus(X, Y, s(s(s(s(0))))).`
    - (c) Write a single rule defining `odd( ?X )` in terms only of `plus`, where `odd( X )` expresses that `X` is an odd number. Test it with some queries.
  3. Write a (recursive) program for `ones_zeros( ?X )` which holds when `X` is a list each of whose members, if any, is either 1 or 0.
  4. Write a program for `hasdups( ?X )` which holds when list `X` contains at least two occurrences of some member. You can use `member/2`, which is a built-in predicate in SWI-Prolog (but not necessarily in other Prolog implementations).
  5. Write a recursive program for `prod(+L, ?P)` which takes a non-empty list `L` of numbers and returns the product `P` of all the numbers in the list. For example, the query `prod( [7, 2, -3], Prod )` returns `Prod = -42`. Refer only to `prod/2` and the built-in predicate `is/2`.

6. The predicate `contains( List1, List2, N )` holds when the list `List2` is a sublist of the list `List1`; the first element of `List2` occupies position `N` in `List1`.

```
contains( [a,b,c,a,d,e,f,g], [a,d,e,f,g], N )
    N = 4
contains( [a,b,c,a,d,c,a,b,c], [a,b], N )
    N = 1
    N = 7
contains( [a,b,c,e,d,f], [c,e,g], N )
    no
```

Define `contains( +List1, +List2, ?N )` in a single rule using `append/3` and `length/2`. Test it with various queries.