Practical Session 6: Prolog

1 Getting started

Prolog is a logic programming language. There are many implementations of Prolog available. We will use SWI-Prolog[1]

Logic programming has 3 kinds of statements: *facts*, *rules* and *queries*.

a finir

2 Facts

Prolog must be fed facts, which are statements which we know are true. They will be the basis of our programs. A fact start with a lowercase and end with a period :

```
prolog_is_simple.
life_is_beautiful.
father(george, maria).
male(george).
```

In the third and fourth lines you see that facts can also have arguments: this particular fact tells us about the link between two terms, i.e. George is Maria's father and George is a male.

Try to write some facts above the following family tree in terms of father/2 and mother/2:

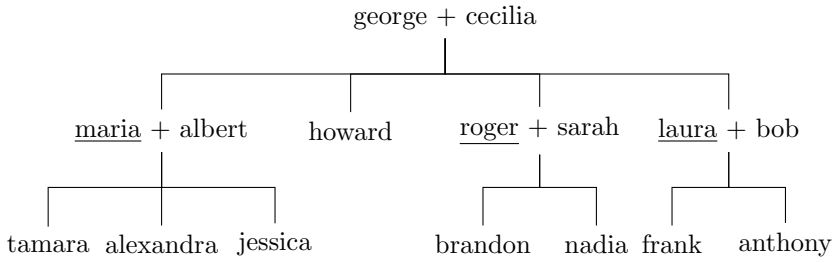
- Maria is Alexandra's mother
- Roger is Nadias's mother
- Laura is Anthony's mother

The provided `family.pl` file lists facts which define this family tree. In the Prolog shell, you can use

```
consult("family.pl").
```

to load these facts into the logic engine.

[1] `http://www.swi-prolog.org/`



3 Logical variables and Unification

In Prolog, objects starting with a capital letter are *logical variables*. These are not variables in the sense you usually mean in other programming languages: they are not a label to some place where you can store data, but they are rather just placeholders for values which are not yet known, like a mathematical variable. Once their value is learnt, it will not change within a clause. A variable can be either free, bound to a term (X = 5.) or aliased to another variable (X = Y.).

Prolog uses pattern matching to define equality. For instance:

```
?- X = X.
true.

?- 9 + 5 = 14.
false.

?- father(george,maria) = father(george,maria).
true.

?- father(george,maria) = father(george,howard).
false
```

No evaluation is performed on the members, they are simply defined as equals if their structures match. Prolog can also *unify* terms: two terms can be unified if it possible to instantiate their variables in a way that both terms are equal. For example:

```
?- father(X,maria) = father(george, Y).
X = george,
Y = maria.
```

The terms father(X, maria) *unifies* with father(george, Y) when X = george and Y = maria. Or

Exercice exemple sur l'unification avec liste infinie (SWI plus intelligent que prolog du livre)

4 Queries

Once you have fed Prolog some facts, you can start asking it some queries. For instance :

```
?- father(george, maria).
true.
```

The ?- denotes the Prolog shell. Of course, you can do more than just querying facts as they were given explicitly. For instance, you could wish to know is Maria's father:

```
?- father(Who, maria).
Who = george.
```

In Prolog a comma , denotes a logical *and* whereas a semi-colon denotes a logical *or*. You can therefore query for all people who are male or are the children of George for instance:

```
?- male(Who); father(george, Who)
```

In this case there are many possible answers: depending on the implementation, you can use the space or tab key to obtain more results and the return key to stop getting results. Prolog has multiple useful built-in predicates to handle these cases, like the **findall** and Prologforall predicates. The **findall** predicate will return all possible answers as a list. For instance, to get all people who are fathers of someone:

```
?- findall(Father, father(Father,X), List).
```

Whereas forall does not provide the list of items the satisfy the query, but rather checks whether all provided items can satisfy the given query or not :

```
?- forall(father(albert,X), mother(maria, X)).
```

Will tell you whether all people who have Albert as father, also have Maria as mother.

Write some queries to :

- know if George is the father of Tamara
- know if Anita is the mother of Brandon
- get all the children of Maria
- get all the sons of Roger
- check that Maria and Albert only have children together
- check that all fathers are male and all mothers are female (without checking this for every known person individually of course)

5 Rules

A *rule* allows us to use known facts to draw conclusions from our world. For example:

```
male(george).                % this is a fact
human(X) :- male(X).         % this is a rule: if X is a male, X is human
```

Would allow you to query for ?− human(george). and obtain **true**.

Write rules to describe the following relationships:

- parent(Parent,Child)
- son(Son,Parent)
- daughter(Daughter,Parent)
- grandfather(Grandfather,Grandchild)
- grandparent(Grandparent,Grandchild)
- brother(Brother,Sibling)
- sibling(Sibling1,Sibling2)
- havechildrentogether(Person1,Person2)
- uncle(Uncle,Person)

Take care that we do not want for someone to be their own brother for instance.

6 Anonymous variables

The underscore character can be used to in place of variables when we are not interested in their value. This character will basically match anything, but each use will be considered a different variable, so you cannot do for instance:

```
grandparent(X, Z) :- parent(X, _), parent(_, Z).
```

because under the hood, each _ will be bound to a new distinct variable called _1, _2, _3 and so on, which means each _ is internally a different variable, like X and Z are different variables.

Use the underscore wildcard to:

- determine whether Laura has one child
- add a human/1 predicate so that *everyone* is a human. Can you also do this without the underscore?
- define a isparent/1 predicate that checks whether a person as at least one child.

7 Recursive rules

It is possible (and powerful) to write recursive rules: rules which reduce the deduction to a base case, like for instance the ancestor rule: someone is an ancestor of someone else if their are their direct parent, of if their are the direct parent of one of their ancestors. But beware that the base case of the recursion must be specified first, lest the logic engine go into an endless loop!

Write a rule to describe the ancestor/2 relationship.

8 Lists

Prolog has a builtin list data structure. They are denoted by square brackets:

```
[a,b,c,d]
```

Similar to LISP lists, they are divided in a head (car) and a tail (cdr), which is itself a list. To handle the head and tail of a list of arbitrary size, you can use the bar operator:

```
[Head|Tail]
```

As in Scheme, the empty list [] forms the base case, a list [a,b,c] can also be written as:

```
[a|[b|[c|[]]]]
```

Translate the following Scheme statements into a Prolog list:

```
(cons a '())
(cons a (cons b (cons c '())))
(cons a L)
(cons a (cons b (cons c L)))
```

Write Prolog clauses to:

- Determine whether an Element belongs to a List

```
member(Element, List)
```

- Append to lists together

```
my_append(List1, List2, Result)
```

- Obtain the last Element of a List

```
last(List, Element)
```

- Shift all elements of a List to the left

```
shiftleft(List, Result)
```

- Shift all elements of a List to the right

```
shiftright(List, Result)
```

- Delete all occurences of Element in a List

```
my_delete(Element, List, Result)
```

Consider the following clauses:

```
h([],[]).
h([X|Y],Z) :- h(Y,L), append(L,[X],Z).
```

1. Try to guess the purpose of these clauses.
2. Try to manually execute h ([1,2,3], L). and h(L ,[1,2,3]).

9 Additional exercise

This exercise is not mandatory.

Write clauses to:

1. Add anElement to a List

   ```
   insert(Element, List, Result)
   ```

2. Give all permutations of a given List

   ```
   permutation(List, Permutation)
   ```

3. Tell whether a Sublist is part of a List

   ```
   sublist(Sublist, List)
   ```