Introduction to Prolog

0 Overview

0.1 Laboratory Guide

Hello and welcome to the Logic Programming laboratory!

This semester, you will learn a new programming language, called Prolog. The solutions to the exercises will be written and tested in a Prolog interpretor (SWISH Prolog).

0.1.1 Laboratory Format

0.1.1.1 Rules

- The laboratory materials must be read **before** the laboratory session.
- At the end of each laboratory session, you must upload your code on Moodle.
- Attendance is mandatory.
- One absence can be recovered (the corresponding assignment MUST be delivered and verified the following week).
- A second and third absence can be recovered (surcharge) in the special laboratory session at the end of the semesterby solving additional assignments (solving the corresponding assignment of the missed session will NOT be considered)
- With more than 3 absence, you can**not** attend the exam in the regular exam session.
- ON OCCASION, you can attend another laboratory session in the same week with another group with the agreement (email or MS Teams) of the teaching assistant.

0.1.1.2 Grading

- The laboratory grade is equal to 30% of the final grade.
- The laboratory grade will be given following a *written test* and an *oral evalution* at the end of the semester, *the activity throughout the semester* will be taken into account.

0.1.2 Laboratory session transfer

If you wish to attend different laboratory sessions, you must follow these rules:

- Student S1 from group G1 can transfer to group G2 if and only if a student S2 from group G2 can be found that is willing to participate with G1 at the assigned hours of G1.
- An email from S1 to both laboratory assistants.
- An email from S2 to both laboratory assistants.

The laboratory session transfer deadline is the end of the second week of the semester.

0.1.3 Resources

Code templates available: https://github.com/ArdeleanRichard/utcn-pl/tree/main/labs-en

Bibliography: https://github.com/ArdeleanRichard/utcn-pl/tree/main/resources/Books

0.2 SWISH

During the laboratory sessions, we will be using the online platform:

https://swish.swi-prolog.org/

0.2.1   SWISH Prolog – online platform (no install required)

SWISH Prolog platform incorporates both the editor and the interpreter and allows the use of both through a simple GUI.

0.2.2   Setup
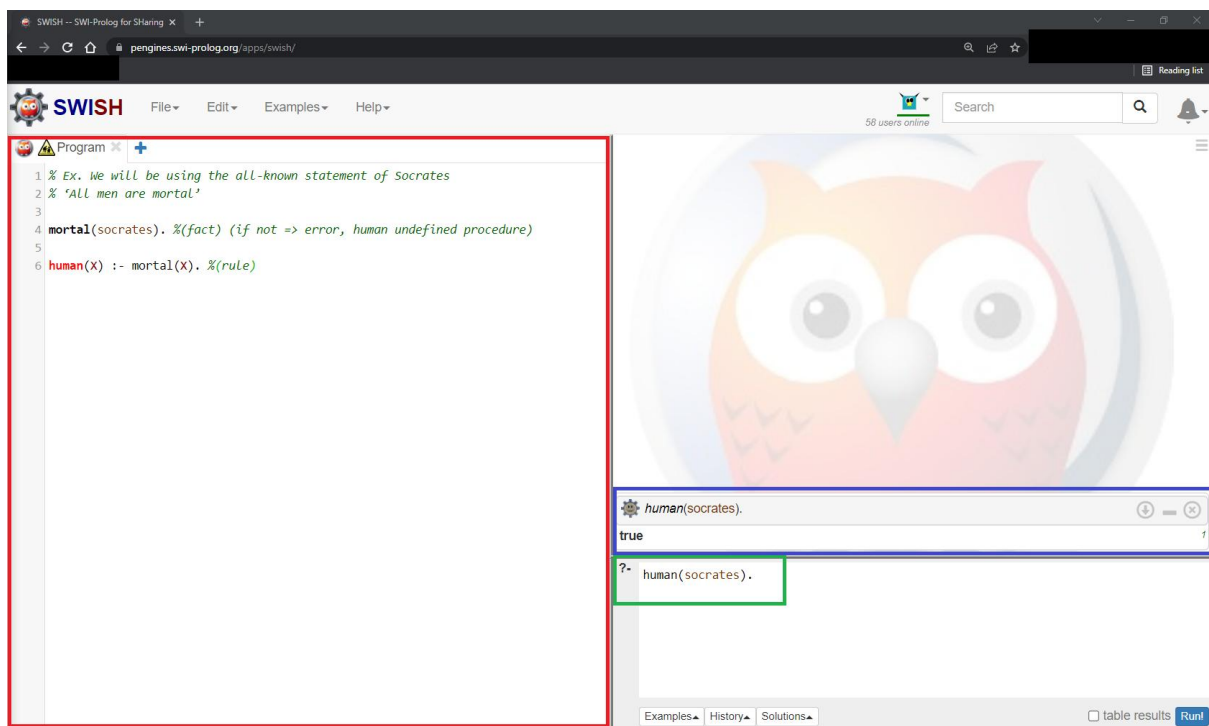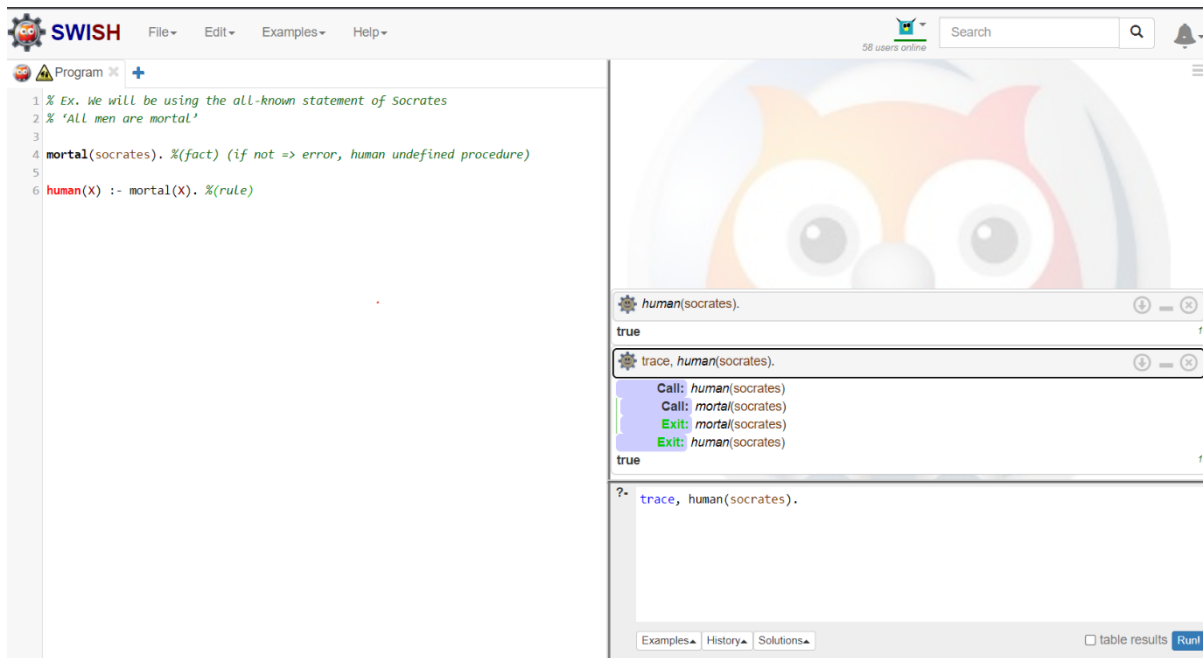

Red – The place where we add the prolog code

Green – The place where we introduce queries

Blue – The place where we receive answers to **queries** based on our **prolog code**

0.2.3   Debugging

Debugging in Prolog is called tracing and "*trace*" is the Prolog command that allows the debugging of Prolog code.

The SWISH version of tracing:

The SWISH version requires the writing of "trace" before the predicate that will be run subsequently.

0.3 Uses of Prolog

Prolog was created in 1972 by Alain Colmerauer and Philippe Roussel, based on the procedural interpretation of Horn clauses. Its popularity picked up in the 80's and 90's throughout Europe, USA and Japan for different systems. It was most used in the development of a particular set of application areas, natural language processing, expert systems (animal care - Auspig, water distribution - WADNES and SERPES, sport advisor - Perfect Pitch) and environmental systems (weather predictions - MM4 Weather Modeling System). The reason for this upsurge was the fast incremental development cycle and rapid prototyping capabilities in the solving of AI problems. Later, its appeal was increased by Object-oriented extensions. Other areas of use include: theorem proving, type systems, automated planning and its originally intended use, natural language processing.

Nowadays, some of the applications of Prolog can be found in (industrial, medical & commercial areas):

- expert systems that solve complex problems without the help of humans (e.g. automatically planning, monitoring, controlling and troubleshooting complex systems)
- decision support systems aiding organizations in decision-making (e.g. decision systems for medical diagnoses)
- online support service for customers

Notably, Prolog has been used in a question-answering computer system capable of answering questions posed in natural language, called Watson (developed by IBM). Specifically, Prolog was used for pattern matching on natural language parse trees.

3

0.4 Why should you learn Prolog?

Learning Prolog is highly beneficial for students due to several reasons. Firstly, it introduces them to the logic programming paradigm, offering a unique perspective on problem-solving distinct from conventional approaches. This fosters a deeper understanding of logic and deductive reasoning, essential skills applicable across various domains. Its support for constraint logic programming further enriches problem-solving abilities, particularly in constraint satisfaction problems. Moreover, Prolog's prominence in artificial intelligence (AI) and expert systems exposes students to cutting-edge technologies and prepares them for roles in AI research and development. Prolog also facilitates ML explainability by providing a transparent and rule-based approach to problem-solving, aiding in understanding and interpreting machine learning models. With Prolog, students embrace a new thought paradigm, where solutions are derived from logical rules and facts rather than explicit instructions. Moreover, Prolog's support for recursion makes it easy to understand and implement recursive algorithms, reinforcing concepts of recursion learned in data structures and algorithms courses. Its interpreter-based nature allows for instantly runnable and testable code, providing immediate feedback and facilitating iterative development. Finally, learning Prolog not only enhances students' problem-solving skills but also offers a recap of data structures and algorithms in a new and engaging paradigm.

1 Theoretic Considerations

In this laboratory session you will get familiar with the principal concepts of the Prolog language: facts, rules and queries. Additionally, the following will be presented: data types used in Prolog and unification rules.

1.1 The Logic Paradigm

A programming paradigm is a fundamental style of programming that dictates:

- The representation of data (ex: facts, variables, classes)
- The preprocessing of data (ex: assignments, comparations, evaluations)

The *Logic Paradigm* belongs to the *Declarative Paradigm*. A declarative programming language answers the „***WHAT*** *should a program do*" question. We will consider the following example in the SQL declarative language:

```
SELECT last_name, first_name FROM Students WHERE year = 3
```

The above instructions do not tell the SQL interpreter how to achieve the search, it only tells the interpreter what conditions the result must respect. Prolog is a logic programming language, therefore it is also a declarative one.

In contrast, an imperative programming language (ex: C, Pascal, C++, C#, Java etc.) answer the „***HOW*** *should a program solve a problem*" question.

1.2 Prolog Concepts

Prolog instructions are represented by *facts* and *rules*. Once the Prolog source program has been consulted by the interpreter, it allows the asking of questions – through *queries*. The answering of questions is determined on the basis of the *facts* and *rules* through the use of the *backtracking* technique.

Anything that is unknown or cannot be proven is considered false.

Comments:

- one line comments start with the **%** symbol
- multiple line comments will be framed by the **/*** and ***/** symbols (similar to comments in the C programming language)

1.3 Data Types

Prolog's single data type is the term. Terms are either: atoms, numbers, variables or compound terms (structures). The figure below presents a classification of the data types in Prolog:
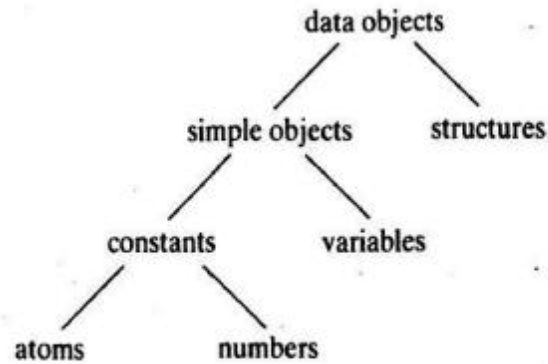
Figure 1 – Prolog data types[1]

Simple:

- constants
  - o numbers (ex: 47, 6.3 etc.)
    - In contrast to C – Prolog does not require a data type declaration as ‚int' or ‚float'
  - o symbols/atoms (ex: girafa, 'Romania', 'antilopa Gnu' etc.)
    - If they contain special characters they need to be surrounded by simple quotation marks
- variables
  - o Start with **Uppercase letter** or with _ (ex: X, Aux, _12 etc.)
  - o A single _ represents a free/anonymous variable

Compound:

- structures (ex: t(1, t(-2, nil, nil), t(8,nil,nil)) etc.)
  - o lists (ex: [], [1, 2, 3], [1, 2 | _] etc.)
    - [] = empty list
    - The list [1,2,3] is internally represented through '.'(1, '.'(2, '.'(3, [])))
    - The template **[H|T]** separates the list in H (= the head of the list) and T (= the tail, **a list** that contains the rest of the elements)
  - o strings (ex: "Hello World")

1.3.1 Exercises on Data Types

1. Which is the nature of the following Prolog terms:

a.	X	d.	hello	g.	[a, b, c]
b.	'X'	e.	Hello	h.	[A, B, C]
c.	_138	f.	'Hello'	i.	[Ana,has,'apples']

2. Look up the following built-in predicates:

- var(Term)

- nonvar(Term)
- number(Term)
- atom(Term)
- atomic(Term)

Note. Prolog defines the *atomic/1* predicate as True if the term is instantiated (i.e. not a variable) and not compound. Thus, atoms and numbers are considered atomic. Moreover, the empty list [] is also considered atomic, as it is the base of the compound structure of the list.

1.3.2   Additional information about the List

Knowing that the template **[H|T]** can be used to separate a list into its head and its tail then (*Remember*: the tail is a list as well):

[1,2,3] can be written as [1 | [2,3]].

Question: Using the template is the way to traverse a list, but what programming technique should be used?

Answer: Recursion. The head is separated from the tail and a recursion is applied on the tail, allowing the processing of the head.

Viewing the template in a recursive fashion, it can again be written as:

[1 | [2 | [3 | []]]]

1.4 Facts

Facts are predicates that are **always** *true*. They are also called axioms (from mathematics). They are allowed to have zero, one or more arguments. The **arity** of a predicate represents the number of arguments of said predicate.

Examples:

```
Code
we_are_in_classroom_108.
animal(elephant).
animal('Gnu antilope').
height(girafa, 5.5). % height in metres
tree( t(1, t(-2, nil, nil), t(8,nil,nil)) ).
```

1.5 Queries

Queries can be viewed as the goals of the Prolog program. The answer of a query can be affirmative or negative. If we use variables in the question we can obtain other information as well.

Examples:

```
Query
?- we_are_in_classroom_108.
true.

?- animal('Gnu antilope').
true.

?- animal(X).
X = elephant; % repeat query with ; or n or space
X = 'Gnu antilope';
false. % there are no other animals defined in the program
```

1.6 Unification

The = symbol realizes the unification between the 2 terms.

Value is any data type that is not / does not contain an uninstantiated variable.

Term 1	Term 2	Operation
Value (/instantiated variable)	Value (/instantiated variable)	Comparation
Value (/instantiated variable)	Uninstantiated variable	Assignment
Uninstantiated variable	Value (/instantiated variable)	Assignment
Uninstantiated variable	Uninstantiated variable	Variables become synonymous (as in pointing to the same location in memory)

Note. Do not confuse the unification of Prolog with the assignment of C.

1.6.1 Exercises on Unification

1. Execute the following unification queries:

a.   ?- a = a.

b.   ?- a = b.

c.   ?- 1 = 2.

8

d.   ?- 'ana' = 'Ana'.

e.   ?- X = 1, Y = X.

f.   ?- X = 3, Y = 2, X = Y.

g.   ?- X = 3, X = Y, Y = 2.

h.   ?- X = ana.

i.   ?- X = ana, Y = 'ana', X = Y.

j.   ?- a(b,c) = a(X,Y).

k.   ?- a(X,c(d,X)) = a(2,c(d,Y)).

l.   ?- a(X,Y) = a(b(c,Y),Z).

m.   ?- tree(left, root, Right) = tree(left, root, tree(a, b, tree(c, d, e))).

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

o.   ?- father(X) = X.

p.   ?- loves(X,X) = loves(marsellus,mia).

q.   ?- [1, 2, 3] = [a, b, c].

r.   ?- [1, 2, 3] = [A, B, C].

s.   ?- [abc, 1, f(x) | L2] = [abc|T].

t.   ?- [abc, 1, f(x) | L2] = [abc, 1, f(x)].

Note1. A list of format [1,2,3] is actually equivalent to [1,2,3 | []] and to

[1 | [2 | [3 | []]]] as well.

Note2. The **[H|T]** template can be extended to [H1, H2 | T] or [H1, H2, H3 |T] or even more. It is important to *remember* that the head cannot be empty, while the tail can be an empty list []. This brings about the implication that a template of type [H1, H2, ..., Hn |T] can only unify with a list of minimum *n* elements. This means that there must be as many stopping conditions as there are heads in order to adress all possible cases.

Note*. (Remember) The unification of Prolog and the assignment of C work in different ways even though they use the same symbol =. While the assignment of C works in a unidirectional manner, the unification of prolog is bidirectional (as the order of the terms does not matter). In some cases it is easier to view it as similar to a comparison (as it even returns a true/false), though this is not an entirely correct view either.

1.7 Rules

The rule represents the definition of a predicate under the form of a clause of Horn type:

p(…) if p11(…) and p12(…) and … and p1n(…).

…

p(…) if pm1(…) and pm2(…) and … and pmk(…).

p(…) = the head of the clause (one predicate at maximum)

p11(…) and … = the body of the clause (zero, one or more predicates)

fact = clause without a body

query = clause without a head

Special symbols (operators) in Prolog:

> **:-** = "if"
>
> **,** = "and"
>
> **;** = "or" (can be achieved – and is equivalent – through the writing of multiple clauses of the same predicate)

Examples:

```
Code
taller(X,Y) :- height(X,Hx), height (Y,Hy), Hx>Hy.
% first we take the height of X, then we take the height of Y
% and finally we compare the heights through the inequality

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Intermediary), path(Intermediary,Y).
% the path between nodes X and Y can be a direct connection between
% the 2 nodes OR if no direct connection exists, we search for a
% indirect connection through an intermediary node
```

1.8 Backtracking

An intrinsic functionality of Prolog is backtracking. Query Answering in Prolog can be viewed (if simplified) as a Depth-First Search (DFS) process. When running a query the interpreter goes into an in-depth search for an answer, if the query is repeated (through ; or Next) the backtracking process starts looking for *another* answer. Remember, it needs to be a distinct answer, otherwise it would result in an infinite loop of the same answer.

Let's implement the following code that determines who can hold a party depending on a list of fanciful requirements:

```
% hold_party/1 is a rule that depends upon the successful execution
% of the birthday/1 and happy/1 facts
hold_party(X):-
   birthday(X),
   happy(X).


% a series of birthday/1 and happy/1 facts
birthday(alex).
birthday(maria).
birthday(adriana).
happy(ana).
happy(george).
happy(adriana).
```

Note1. Remember "/x" represents the arity of a predicate = the number of parameters.

Note2. In general, we want facts to be specific -> as you can see, they contain constants. Simultaneously, we want rules to be general -> they use a variable. Why? When we query a rule, we want to be able to let it find an answer however vague the question is. If the rule would be specific, we would only find one answer.

We will be following the execution of the *hold_party/1* predicate through SWISH Prolog:

Follow the execution of:

```
Query
?- trace, hold_party(Who).
```

1.9 Recursion

Let's empirically separate recursion into a subset of fundamental parts, we will be using the C implementation of factorial:

1. Any recursion requires, by its definition, the call of the function within the function.
2. It also contains a step, in the case of the factorial, the step is a decrement (or increment).
3. Then, there's the processing – for the factorial, this is the multiplication – this can be overlooked for now.
4. Let's follow what happens when we try to compute the factorial of 3.
   a. Multiply by 3 & decrement -> recursion
   b. Multiply by 2 & decrement -> recursion
   c. Multiply by 1 & decrement -> recursion
   d. Multiply by 0 & decrement -> recursion
   e. Multiply by -1 & decrement -> recursion
   f. ...
   g. Overflow – Multiply by 2147483647 & decrement -> recursion
   h. *Do you see the problem?* Ignoring the fact that there's a multiplication by 0, it's an infinite loop. An integral part of a recursion call is the stopping condition.

We will be taking the example of a student that has been at home after the exam session and is trying to get back to school.

```
Code
% the on_route/1 fact
on_route(dorms).
% the on_route/1 rule – this is a recursive rule
on_route(Place):-
   move(Place, Method, NewPlace),
   on_route(NewPlace).


% the move/3 facts
move(home, taxi, trainstation).
move(trainstation, train, cluj).
move(cluj, bus, dorms).
```

Follow the execution of:

```
Query
?- trace, on_route(home).
```

2 Final Exercise
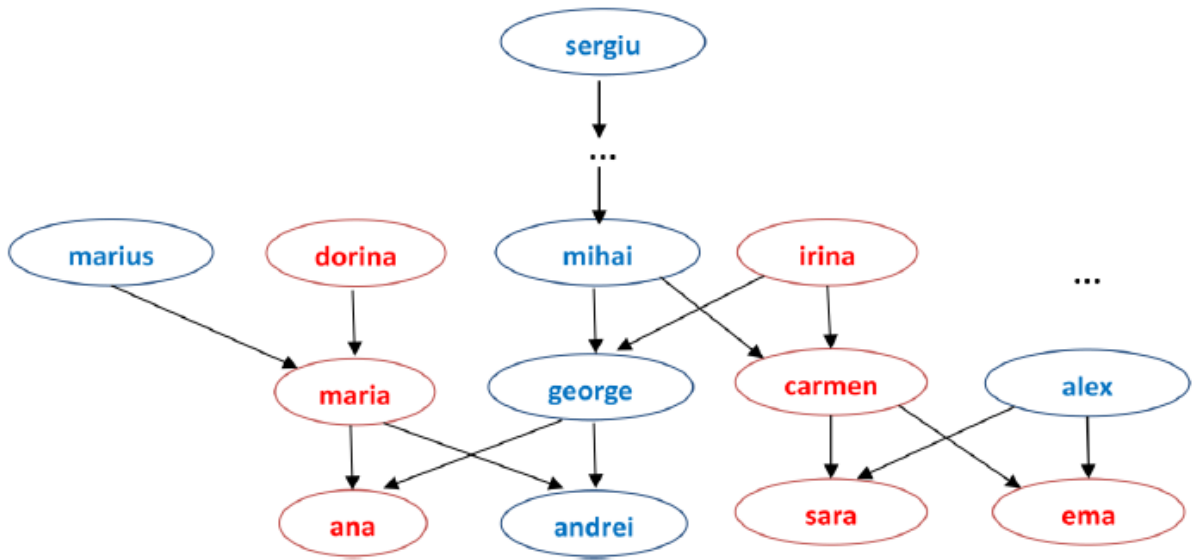
1. Write new predicates for the kinship relations.

Figure 1. Family tree

```
Code
% The woman/1 predicate – remember /x specifies the arity as ‚x'
woman(ana).
woman(sara).
woman(ema).
woman(maria).
% … add the remaining facts of this predicate


% The man/1 predicate
man(andrei).
man(george).
man(alex).
% … add the remaining facts of this predicate


% The parent/2 predicate
parent(maria, ana). % maria is the parent of ana
parent(george, ana). % george is the parent of ana
parent(maria, andrei).
parent(george, andrei).
% … add the remaining facts of this predicate
```

```
% The mother/2 predicate – based on the parent and woman predicates
% X is the mother of Y, if X is a woman and X is the parent of Y
mother(X,Y) :- woman(X), parent(X,Y).
```

1.1.　Test the following queries:

Do remember: queries are preceded by the '**?-**' operator, *which is already written*, the rest of the lines represent the answers of Prolog to the queries):

```
Query
?- man(george).      % is george a man?
true.

?- man(X).          % who is a man?
X = andrei ? ;               % we can repeat the question using ; or n or space
X = george ? ;
X = alex.

?- parent(X, andrei). % Who are the parents of andrei?
X = maria ? ;
X = george.

?- parent(maria, X). % Who are the children of maria?
X = ana ? ;
X = andrei.

?- mother(ana, X).  % Who are the children of ana?
false.

?- mother(X, ana).  % Who is the mother of ana?
X = maria ? ;       % repeat question: Does ana have another mother?
false.
```

1.2.　Write the father/2 predicate.
1.3.　Complete the man/1, woman/1 and parent/2 predicates to cover the whole genealogic tree from above.

1.4.　Test the following queries:

```
?- trace, father(alex, X).
?- trace, father(X, Y).
?- trace, mother(dorina, maria).
```

1.5. Test the following predicates:

Code

```
% The sibling/2 predicate
% X and Y are siblings if they have at least one parent in common
% and X is different from Y
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\=Y.


% The sister/2 predicate
% X is the sister of Y if X is a woman and X and Y are siblings
sister(X,Y) :- sibling(X,Y), woman(X).


% The aunt/2 predicate
% X is the aunt of Y if she(X) is the sister of Z and Z is the parent of Y
aunt(X,Y) :- sister(X,Z), parent(Z,Y).
```

1.6. Write the brother/2, uncle/2, grandmother/2 and grandfather/2 predicates.

1.7. Follow the steps of finding the answer of the following queries (through the use of *trace*):

Query

```
?- trace, aunt(carmen, X).
?- trace, grandmother(dorina, Y).
?- trace, grandfather(X, ana).
```

1.8. Write the ancestor/2 predicate. X is the ancestor of Y if X is linked to Y through a series (regardless of number) of parent relationships.