PROLOG

24 de fevereiro de 2019 12:29

Cátia Santos @2020

Prolog is a **logic** programming language.

- Is intended primarily as a **declarative** programming language.
- Is a **collection** of **facts and rules** that can be **queried**, focused on describing facts and relationships about problems.

Facts are what is known.

Rules are used when you want to say that a **fact depends on a group of facts**

- **Facts** needs to be **defined** first, before being used in a **rule.**
- If, in one fact, we say **one thing is true(**for example what is behind the if), **the rest is also true**(what is after the if) and the other way around.

AND	,
IF	:-
OR	;
NOT	not

The facts and rules are **clauses** and are stored in a file called a **Database** or **Knowledge Base**.

Commands inside Prolog are called **predicates**.

We should keep **predicates** of the same type **organized**(grouped) in out Database.

A **Variable** is an object we can't name at the time of execution

- They written in are uppercase, and begin either with an **uppercase** letter or _ , it can contain letters, numbers, +, -, _, *, /, <, >, :, ., ~,
- An **instantiated** variable is one that stands for an object.
- If the **same variable** name is used in **2** different **questions**, it represents 2 different variables.
- An **uninstantiated** variable can be used to search for any match.
- We can also use variables in the **database**
  - The **singleton warning** means you defined a variable that you didn't do anything with.
- We can use sometimes an **anonymous variable**, if we don't intend to use the variable more than once. Example : (male(_).)
  - We can also use it when we don't want a value returned.

An **atom** is a constant, the argument(s) to the predicate-

- They are written in lowercase and it can contain letters, numbers, +, -, _, *, /, <, >, :, ., ~, & but it cannot start by _

Complex terms and Structures -> A **Structure** is an object made up from many other objects (**components**)

- Structures allow us to add **context** about what an **object** is.
- They have a **functor** followed by a list of arguments.
- The number of arguments of a structure it's called an **arity**.

How to load a Knowledge Base:

- [knowledge].
- consult('knowledge.pl').

halt. -> exits the Prolog system.

listing. -> Displays the contents of the database

Resources:

- Prolog Official:
  - http://www.learnprolognow.org/
- Learn Prolog in One Video:
  - http://www.learnprolognow.org/ ❤️
- Prolog Tutorial:
  - Prolog - Introduction

SWI Prolog (Windows):
https://www.swi-prolog.org/

gprolog(Windows):
http://www.gprolog.org/#download

Homebrew (Linux):
https://brew.sh/

The Art of Prolog,...

Example of how to **output information**:

- write('Hello World'),nl,write('Let \'s Program').

write -> Prints text between quotes to the screen
nl -> stands for new line and
\ -> allows you to use quotes

Example of how to **create a Fact:**
<relationship>(<object>,<object>).

- loves (romeo, juliet).

loves -> **predicate**
romeo,juliet -> **atoms**(constants) and the **predicate arguments**

Example of how to **create a Rule:**
<relationship>(object) :- <relationship>(object).

- loves(juliet, romeo) :- loves(romeo, juliet).

:- -> **if**
If the item on the right is true, then so is the item on the left
To check the evaluation above:
| ?- loves(romeo,X).
X = juliet
yes

Example of how to **define a new predicate**:
does_alice_dance :- dances(alice),
write('When Alice is happy and with

halt. -> exits the Prolog system.

listing. -> Displays the contents of the database

change_directory('pathname'). -> Changes directory.

Learn by examples

Example:

```
male(albert).
male(bob).
male(bill).
male(carl).
male(charlie).
male(dan).
male(edward).
female(alice).
female(betsy).
female(diana).
```

female(alice). = yes -> to find out if Alice is a woman.
listing(male). -> list all clauses defining the predicate male
male(X), female(Y). -> shows all combinations of male and female
use **;** to cycle through the options.
When you are cycling through the results the **no** at the end **signals** that there are **no more results**
X -> stands for a **variable**
female(X). -> returns all females

Example:

```
happy(albert).
happy(alice).
happy(bob).
happy(bill).
with_albert(alice).
parent(albert, bob).
parent(albert, betsy).
parent(albert, bill).
parent(alice, bob).
parent(alice, betsy).
parent(alice, bill).
parent(bob, carl).
parent(bob, charlie).
```

We can for example, define a new **fact**, saying that if Albert is happy, he runs.

```
runs(albert) :-
        happy(albert).
```

We can **input more than one condition** on a **rule**, using the comma(**,**).
Comma stands for **and**

```
dances(alice) :-
  happy(alice),
  with_albert(alice).
```

Is also possible to define the 2 rules separately for the same fact.

```
dances(alice) :-
    happy(alice).

dances(alice) :-
    with_albert(alice).
```

predicate:

```
does_alice_dance :- dances(alice),
    write('When Alice is happy and with Albert she dances').
```

In the terminal we can use the command:

```
| ?- does_alice_dance.
```

Result:

```
When Alice is happy and with Albert she dances
yes
```

Format

Use format to get the results

- ~w represents where to put each value in the list at the end
- ~n is a newline
- ~s is used to input strings
- ~2f is used to show floats with two decimal digits.

Example custom predicate:

```
get_grandparent :-
        parent(X,carl),
        parent(X,charlie),
        format('~w ~s grandparent ~n',
        [X, "is the"]).
```

Result:

```
bob is the grandparent
```

We can also give **arguments** to a custom predicate:

```
grand_parent(X, Y) :-
  parent(Z, X),
  parent(Y, Z).
```

We can then call the predicate like:

```
grand_parent(carl, A).
```

How to create a **structure**:

has(albert,olive). -> without structure

owns(albert,pet(cat,olive)). -> with structure

We can also, for example define the meaning of being **vertical** and being **horizontal**:

```
vertical(line(point(X, Y), point(X, Y2))).
horizontal(line(point(X, Y), point(X2, Y))).
```

Arithmetic operations are allowed, and equal is represented by **is**.
Prolog provides 'is' to evaluate mathematical expressions
How to perform **comparisons**:

happy(alice).

```
dances(alice) :-
    with_albert(alice).
```

We can perform a question with **more** than one **predicate**:

```
parent(X, bob), -> is bob's parent
   dances(X). -> also dances
parent of  bob that dances
 X = alice ? ;
no
```

And also with **more** than one **variable**:

```
parent(albert, X), -> is albert a parent
    parent(X, Y).  -> does his children have any children
X = bob
Y = carl ?
X = bob
Y = charlie ? ;
no
```

We can then create a **custom predicate** on this situation, to be easier:

```
get_grandchild :-
        parent(albert, X), -> is albert a parent
        parent(X, Y), -> does his children have any children
        write('Alberts grandchild is '),
        write(Y), nl.
```

We can now check in the terminal:

```
| ?- get_grandchild.
grandchildren of albert
Alberts grandchild is carl
true ? ;
Alberts grandchild is charlie
true ? ;
no
```

If is not used in Prolog, we use different predicates for different situations, similar to a **case** operation.

Example:

```
what grade(5) :-
        write('Go to kindergarten').
what grade(6) :-
        write('Go to 1stGrade').
what grade(Other) :- -> any other value
        Grade is Other -5,  -> argument -5
        format('Go to grade ~w', [Grade]).
```

Use:

```
what_grade(5).
```

Result:

```
Go to kindergarten
```

Example:

```
warm_blooded(penguin).
warm_blooded(human).
produce_milk(penguin).
produce_milk(human).
have_feathers(penguin).
have_hair(human).
mammal(X) :-
```

Prolog provides 'is' to evaluate mathematical expressions

How to perform **comparisons**:

- =

Example:

```
alice = alice.
yes
 'alice' = alice.
yes
```

- \+ **Not equal**

Example:

```
\+ (alice = albert).
yes
```

- > , >= , =< , <

Example:

```
5 > 2
yes
```

- =:= **Equality between expressions**
- =\= **Inequality between expressions**

Example:

```
5+4 =:= 4+5.
yes
```

- ; **Or is true if one or the other is true**

Example:

```
5 > 10 ; 10 < 100.
yes
```

- Also we can for example check if we can assign a value to a variable

Example:

```
W = alice.
yes
```

Mathematical Operations:

- + , -, * , /
- You can also use **parenthesis** ().
- **mod** -> Modulus
- **random(X,Y,V)** -> Generate random values between X and Y
- **between(X,Y,V)** -> Get all values between X and Y
- **succ(X,V)** -> Increments a value to X and assigns it to V
- **abs** -> Get na absolute value
- **max** -> Gets the largest of the values
- **min** -> Gets the smallest of the values.
- **//** -> Divides while disregarding decimals
- round, truncate, floor, ceiling
- sqrt, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh,
- asinh, acosh, atanh, log, log10, exp, pi, e, etc…

```
have_feathers(penguin).
have_hair(human).
mammal(X) :-
  warm_blooded(X),
  produce_milk(X),
  have_hair(X).
```

We can use **trace** to see how Prolog evaluates queries one at a time, since it activates the debugger mode.

trace. -> Turns on trace

notrace. -> Turns off trace

After being activated we can then for example query for mammals and see how everything is processed:

```
{trace}
mammal(human).
    1  1  Call: mammal(human) ?
    2  2  Call: warm_blooded(human) ?
    2  2  Exit: warm_blooded(human) ?
    3  2  Call: produce_milk(human) ?
    3  2  Exit: produce_milk(human) ?
    4  2  Call: have_hair(human) ?
    4  2  Exit: have_hair(human) ?
    1  1  Exit: mammal(human) ?
(1ms) yes
{trace}
```

Example:

```
parent(albert, bob).
parent(albert, betsy).
parent(albert, bill).
parent(alice, bob).
parent(alice, betsy).
parent(alice, bill).
parent(bob, carl).
parent(bob, charlie).
```

Recursion cycles through possible results until related returns a true

```
related(X, Y) :-
  parent(X, Z),
  related(Z, Y).
```

How to **write to a file**:
In order to write to a file, you need to start by defining the file, then the text to write, and open a connection to the file, which is called a **stream**.

Example:

```
write_to_file(File, Text) :-
  open(File, write, Stream),
  write(Stream, Text), nl,
  close(Stream).
```

How to **read from a file:**

Example:

```
read_file(File) :-
    open(File, read, Stream)
    get_char(Stream, Char1),   -> outputs the characters
    process_stream(Char1, Stream), -> continues getting the characters until the end of the file.
    close(Stream).
```

atan, atan2, sinh, cosh, tanh,

- asinh, acosh, atanh, log, log10, exp, pi, e, etc…
- **%** -> percentage

How to **output** a message:

- **write** -> outputs whatever message is inside quotes

Example:

```
write('Test123'),nl.
Test123
```

- **writeq** -> outputs whatever message is inside the parenthesis(quotes included)

Example:

```
write('Hello'),nl.
'Hello'
```

- **writeln** -> outputs the message + a new line

Example:

```
writeln('Test values').
```

- **writef** -> outputs the message allowing formatted content.

Example:

```
writef('Test values %w.'\r\n, [List]).
```

How to get **input** from:

- **read** -> read input form the user

Example:

```
say_hi :-
  write('What is your name? '),
  read(X),
  write('Your name is').
  write(X).
```

Result:

```
say_hi.
What is your name? 'Cat'.
Your name is Cat
```

- **get/put** -> receives one character(ASCII value)

Example:

```
fav_char :-
   write('What is your fav character? '),
   get(X),
   format('The ASCII value ~w is ', [X]),
   put(X), nl.
```

Result:

```
fav_char.
What is your fav character? t
The ASCII value 116 is t
```

How to create a **loop**:
The technique previous seen of **recursion** is what we use to create a

process_stream(Char1, Stream), -> continues getting the characters until the end of the file.
close(Stream).

process_stream(end_of_file, _) :- !. -> **!** or **cut** is used to end backtracking or this execution

```
process_stream(Char, Stream) :-
    write(Char),
    get_char(Stream, Char2),
    process_stream(Char2, Stream).
```

Result:
write_to_file('test1.txt, 'Random String').c

read_file('test1.txt').
Random String

Any predicate can be changed during the execution of the program, but in order to do so, they need to be marked as **dynamic** beforehand.
:- dynamic(predicate/attribute number).

Example:
```
:- dynamic(father/2).
:- dynamic(likes/2).
:- dynamic(friend/2).
:- dynamic(stabs/3).
father(lord_montague,romeo).
father(lord_capulet,juliet).
likes(mercutio,dancing).
likes(benvolio,dancing).
likes(romeo,dancing).
likes(romeo,juliet).
likes(juliet,romeo).
likes(juliet,dancing).
friend(romeo,mercutio).
friend(romeo,benvolio).
stabs(tybalt,mercutio,sword).
stabs(romeo,tybalt,sword).
```

- **assertz** -> adds a new clause at the end of the list(database)

Example:
assertz(friend(benvolio, mercutio)).

- **asserta** -> adds a new clause at the beginning of the list(database)

Example:
asserta(friend(benvolio, mercutio)).

- **retract** -> deletes a cause from the list

Example:
retract(likes(mercutio,dancing)).

- **retractall** -> deletes all causes that match a criteria

Example:
retractall(father(_,_)).

Example 2:
retractall(likes(_,dancing)).

How to create a **loop**:
The technique previous seen of **recursion** is what we use to create a loop.

Example:
```
count_to_10(10) :- write(10), nl.
count_to_10(X) :-
  write(X),nl,
  Y is X + 1, -> increments one to the X value
  count_to_10(Y). -> calls the same predicate for a new argument
```

Example 2:
```
count_down(Low, High) :-  -> Assigns values between Low and High to Y
  between(Low, High, Y), -> Assigns the difference to Z
  Z is High - Y,
  write(Z),nl,
```

Strings can be manipulated according with our needs

- **name** -> converts a string into a series of Ascii characters

Example:
name('A random string', X).

Result:
X = [65,32,114,97,110,100,111,109,32,115,116,114,105,110,103]

We can also use it the other way around:
Example 2:
name (X, [65,32,114,97,110,100,111,109,32,115,116,114,105,110,103]).

Result:
X = A random string

Relating two lists:
You can use **maplist** to relate two lists together.

Example:
maplist(Result, List1, List2).

Search a variable for it's elements:
In order to find a certain term within a variable, we can use the **findall** function.

Example:
findall(Variable, Term, Result).

Get **totals** from lists
There are several ways we can get a

Is possible to store atoms, complex terms, variables, numbers and others lists in a **list**.

In Prolog we use lists to store data that has **an unknown number of elements**.

Use a **list constructor** to add values to a list:
Example:
write([albert|[alice, bob]]), nl. -> adds albert to the list

- **length** -> gets the length of a list

Example:
length([1,2,3], X).

- We can **divide** a list into its **head** and **tail** with **|**

Example:
[H|T] = [a,b,c].

- We can use **|** to access values of lists in lists

Example:
[_, _, [X|Y], _, Z|T] = [a, b, [c, d, e], f, g, h].

- **member** -> finds out if a value is in a list with member

Example:
member(a, List1).

Example 2:
member(X, [a, b, c, d]).

- **reverse** -> reverses a list

Example:
reverse([1,2,3,4,5], X).

- **append** -> concatenate 2 lists together

Example:
append([1,2,3], [4,5,6], X).

Example on how to output the items in a list on separate lines:

```
write_list([Head|Tail]) :-
  write(Head), nl,
  write_list(Tail).
```

Result:

```
write_list([1,2,3,4,5]).
1
2
3
4
5
```

Usefull operations with Lists:

```
count(0, []).
count(Count, [Head|Tail]) :-
        count(TailCount, Tail), Count is TailCount + 1.

sum(0, []).
sum(Total, [Head|Tail]) :-
        sum(Sum, Tail), Total is Head + Sum.
```

Get **totals** from lists
There are several ways we can get a total from a list, one of them is using the **sum_list** function:

Example:
sum_list(List, Result).

You can also do a **sum** like:
Example:
sum(List,Result).

Or to sum a value from inside a list

```
sum( [], 0).
sum( [mensal(_,D)|T], Sum ) :-
   sum( T, DT ),
   Sum is DT + D.
```

How to **count** the number of times a predicate is true:

Example:

```
count(P,Count) :-
      findall(1,P,L),
      length(L,Count).
```

If you want the value aggregated is better to use **aggregate_all**:

Example:

```
aggregate_all(count,
whatwewanttofind, Variable),
```

How to **sort** your lists:

Example:
keysort(List, ResultList).

And in order to **reverse** the list

Example:
reverse(List.ResultList).

Other functions:
bagof

```
append(List1, List2, List12)
member(Element, List)
reverse(List1, List2)
delete(List1, Element, List2)
select(Element, List1, List2)
permutation(List1, List2)
prefix(Prefix, List)
suffix(Suffix, List)
sublist(List1, List2)
last(List, Element)
length(List, Length)
nth(N, List, Element)
min_list(List, Min)
max_list(List, Max)
sum_list(List, Sum)
sort(List1, List2)
```

```
sum(0, []).
sum(Total, [Head|Tail]) :-
        sum(Sum, Tail), Total is Head + Sum.

average(Average, List) :-
        sum(Sum, List),
        count(Count, List),
        Average is Sum/Count.
```

```
sum_list(List, Sum)
sort(List1, List2)
```