Lab 5: Prolog Fundamentals - Facts, Rules and Queries

CSI 3120 A - Programming Language Concepts

Fall 2024

School of Electrical Engineering and Computer Science
University of Ottawa

Anthony Le - 300287511
Yaroslav Volkov - 300235591

Experiment Date: Oct 18, 2024
Submission Date: Oct 24, 2024

Lab Tasks and Their Desired Outputs

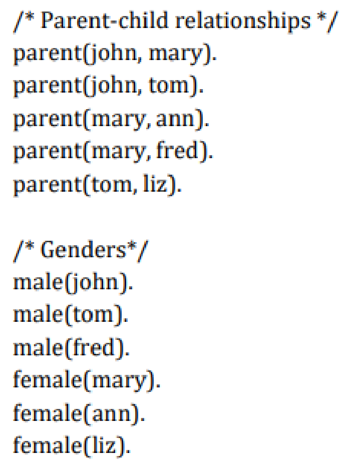
In this lab, students will:

- Implement a series of prolog tasks using Prolog and the fundamental aspects regarding the language.
- Understand the different prolog fundamentals used for these tasks:
    - Defining conditional predicates to figure out the relations for the already-set rules.
    - Recursion and array iteration in Prolog for filtering.
    - Puzzle solving by utilizing built-in prolog functions and coded predicates.

Task A: Family Tree.

In this task, we are tasked with defining predicates to determine additional rules for a small family tree to discover the deeper relationships within the family tree.

With the given set of rules, we are tasked with determining predicates to determine a sibling, grandparent, and ancestor.



```
/* Parent-child relationships */
parent(john, mary).
parent(john, tom).
parent(mary, ann).
parent(mary, fred).
parent(tom, liz).

/* Genders*/
male(john).
male(tom).
male(fred).
female(mary).
female(ann).
female(liz).
```

Figure 1. List of given rules to define the family tree for lab5 Task A.

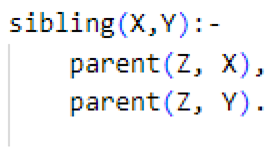
Step 1. Defining `sibling/2`

This predicate was used to define siblings, as individuals sharing a parent between each other while not being the same person.

This implementation was done through the following steps:

- We implement a conditional sibling predicate that depends on separate conditions to be true. If they are true, then the following inputs are siblings:

  - We first check to see if the first input (X) is a part of a parent rule, where the input would be the second argument (in parent(Z, X). We Instantiate the supposed parent to a variable 'Z', which will then be used in the second conditional rule.
- The second conditional rule is the same as the first rule, except that we are now checking for the second input (parent(Z, Y)). With the 'Z' variable, with the 'Z' variable being the same, this checks if the parent is the same.

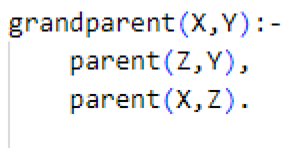


```
sibling(X,Y):-
    parent(Z, X),
    parent(Z, Y).
```

Figure 2. Code Snippet for Sibling rule for lab 5 Task A

Step 2. Defining `grandparent/2`

This predicate is used to determine whether one variable is the grandparent of another; this is used with the similar logic towards the sibling rule, except it is manipulated to check the proper family tree structure.

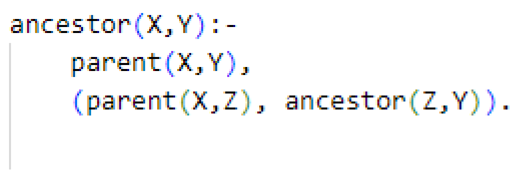


```
grandparent(X,Y):-
    parent(Z,Y),
    parent(X,Z).
```

Figure 3. Code Snippet for Grandparent Rule for Lab 5 Task A.

Step 3. Defining `ancestor/2`

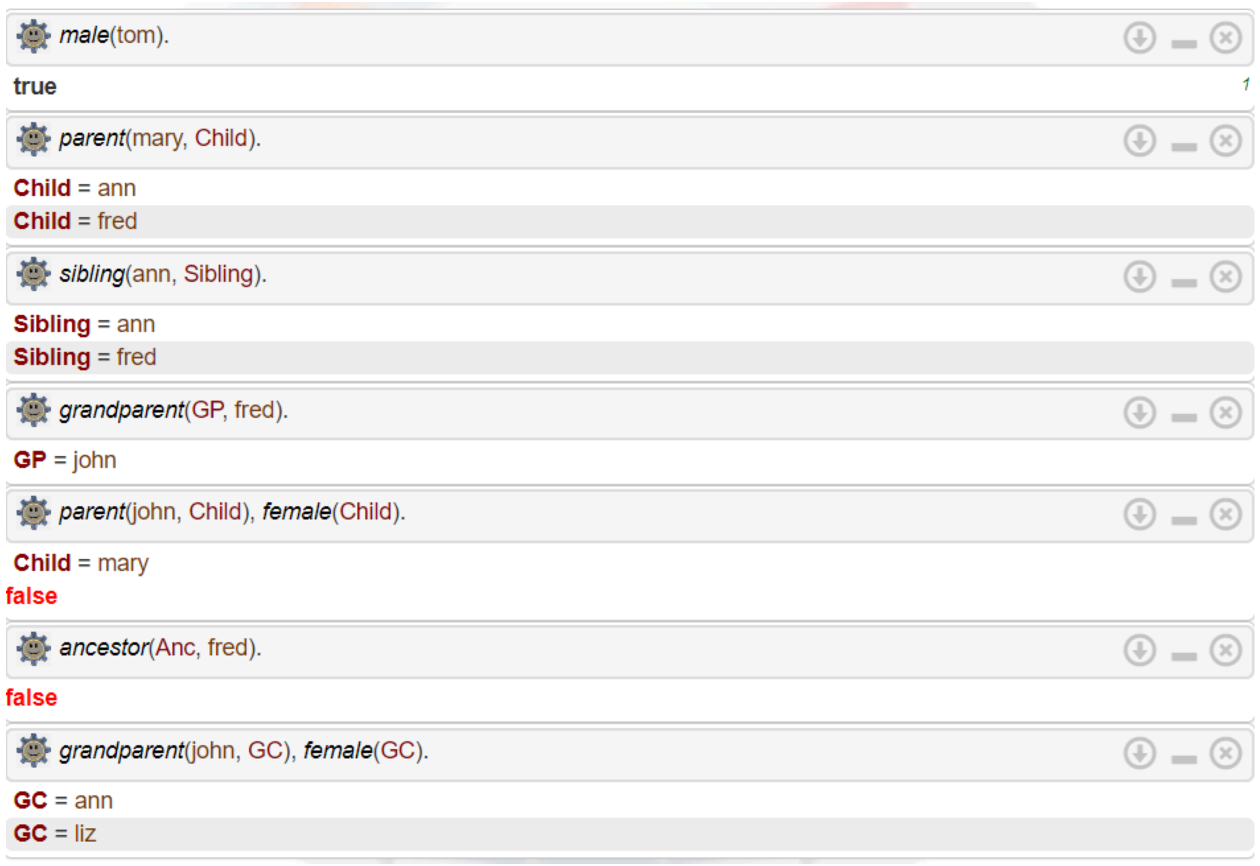
Following the logic of what an ancestor is, we defined a rule to determine the oldest member found within the family tree, which should be equal to the variable 'X'.



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

Figure 4. Code Snippet for Ancestor Rule for Lab 5 Task A.

Desired Inputs and Viewed Outputs



```
male(tom).
true
parent(mary, Child).
Child = ann
Child = fred
sibling(ann, Sibling).
Sibling = ann
Sibling = fred
grandparent(GP, fred).
GP = john
parent(john, Child), female(Child).
Child = mary
false
ancestor(Anc, fred).
false
grandparent(john, GC), female(GC).
GC = ann
GC = liz
```

Figure 5. Inputs and Resulting Outputs for Lab 5 Task A.

These outputs did not entirely meet the expectations, as the `ancestor/2` function is not entirely accurate and does not fully reflect the proper implementation of the ancestor predicate.

Task B: Recursive List Processing.

We were tasked with creating the predicate `sum_odd_numbers/2` to take in a list and calculate the sum of all odd numbers inside the list recursively.

This function was made using a helper function with a base case and normal case in order to properly cover all possible outcomes of the function.

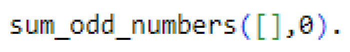
Step 1. Defining the Base Case.

Before coding the whole function, we coded a base case in order to cover what would happen if the list were to be empty.

This was done by doing the following:

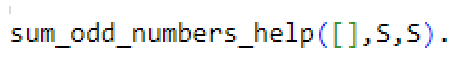
- If the list was empty, the function `sum_odd_numbers/2` would simply return the value of S to 0;

    - This way, if the list would be empty, it would simply return the value of 0 since no value was added.
- This was also implemented in the helper function, where instead when the list becomes empty, the Accumulator variable that is used as a helper to count the accumulator would be equal to the sum to return the value of S.



```
sum_odd_numbers([],0).
```

Figure 6. Code Snippet for Base Case of Main Function for Lab 5 Task B.



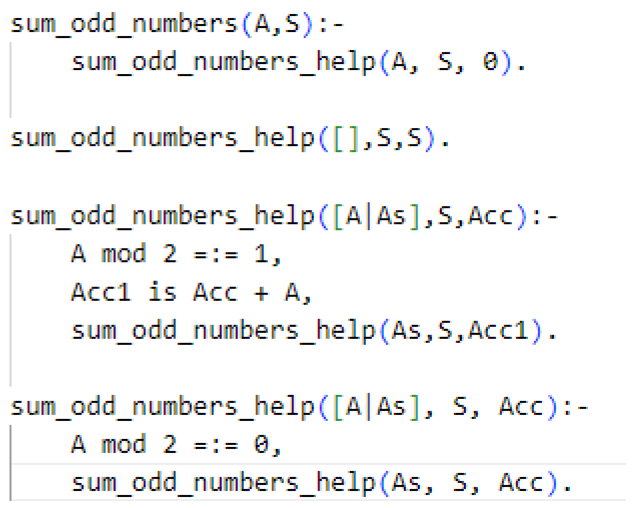
```
sum_odd_numbers_help([],S,S).
```

Figure 7. Code Snippet for Base Case of Helper Function for Lab 6 Task B.

Step 2 and 3. Defining the Recursive Case and Skipping Even Numbers.

For this part of the task, the recursive aspect is mainly focused on the helper function, where then we can control the behavior of each of the values within the list that are odd or even.

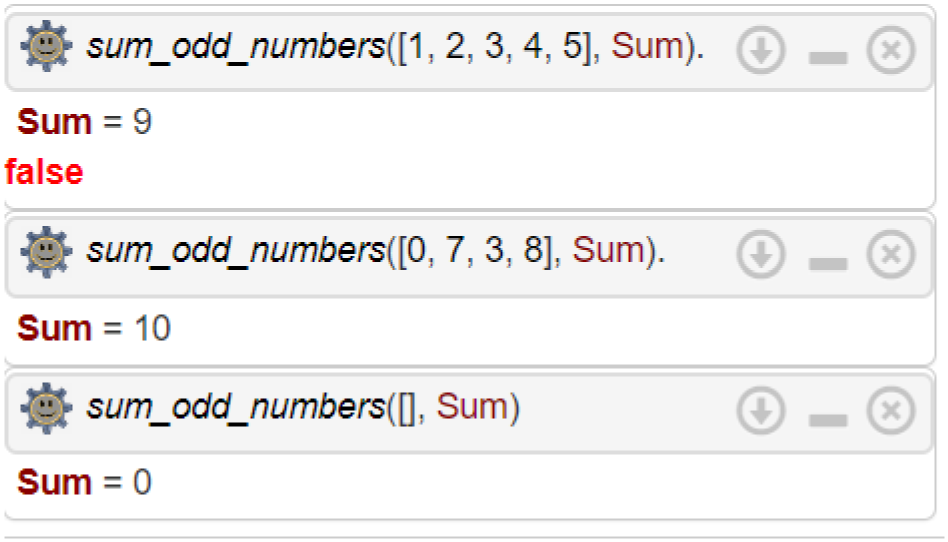
- If the value within the list is odd, meaning it passes the condition, we simply create a temporary variable that adds the value to the Accumulator's value, which that temporary variable is then used in the next recursive function.
    - The odd check is done by using a modulo 2 check; if A modulo 2 returns a value of 1, then we would know that the value inside the list is odd, and thus we move on to add it to the accumulator variable
- If the value is even, meaning it does not pass the condition, we simply created another predicate that handles even numbers, where if it doesn't pass the modulo 2 check, it will simply continue to iterate through the list by simply calling the function with the tail end of the list.



```
sum_odd_numbers(A,S):-
    sum_odd_numbers_help(A, S, 0).

sum_odd_numbers_help([],S,S).

sum_odd_numbers_help([A|As],S,Acc):-
    A mod 2 =:= 1,
    Acc1 is Acc + A,
    sum_odd_numbers_help(As,S,Acc1).

sum_odd_numbers_help([A|As], S, Acc):-
    A mod 2 =:= 0,
    sum_odd_numbers_help(As, S, Acc).
```

Figure 8. Code Snippet for Recursive Behavior and Modulo 2 condition check for Lab 5 Task B.

Inputs and Desired Outputs.



```
sum_odd_numbers([1, 2, 3, 4, 5], Sum).
Sum = 9
false

sum_odd_numbers([0, 7, 3, 8], Sum).
Sum = 10

sum_odd_numbers([], Sum)
Sum = 0
```

Figure 9. Inputs and Desired Outputs for Lab 5 Task B.

These outputs meet our expectations for the following code.

Task C: Hidden Treasure Puzzle.

This task requires us to solve the puzzle by listing out the house colours in its possible combination that matches with all four predicates.

Step 1. Permutation of the possible colours.

To find the multiple combinations that are possible as an answer, we simply used the built-in permutation functions to get the possible colours based off the possible colours that will be seen within the array.

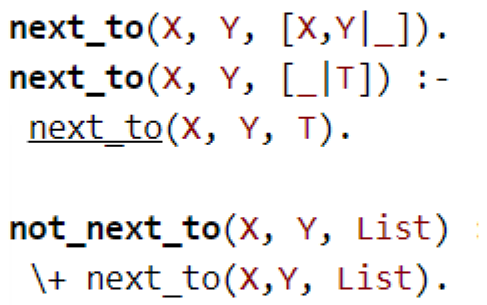
```
Houses = [_,_,_,_],
permutation(Houses, [red, blue, green, yellow]),
```

Figure 10. Code Snippet of the Permutation Function To Create Possible Combinations for Lab 5 Task C.

Step 2. Defining the `Next_to/3` and `not_next_to/3` helper functions.

These helper functions are used to ensure that the house certain colours are beside each other baked off the clues. This was done by recursively checking whether the two colours, `X` and `Y` are beside each other, which stops the function if they are. If not, then the next_to function is called recursively to continue through the list until the condition is met.

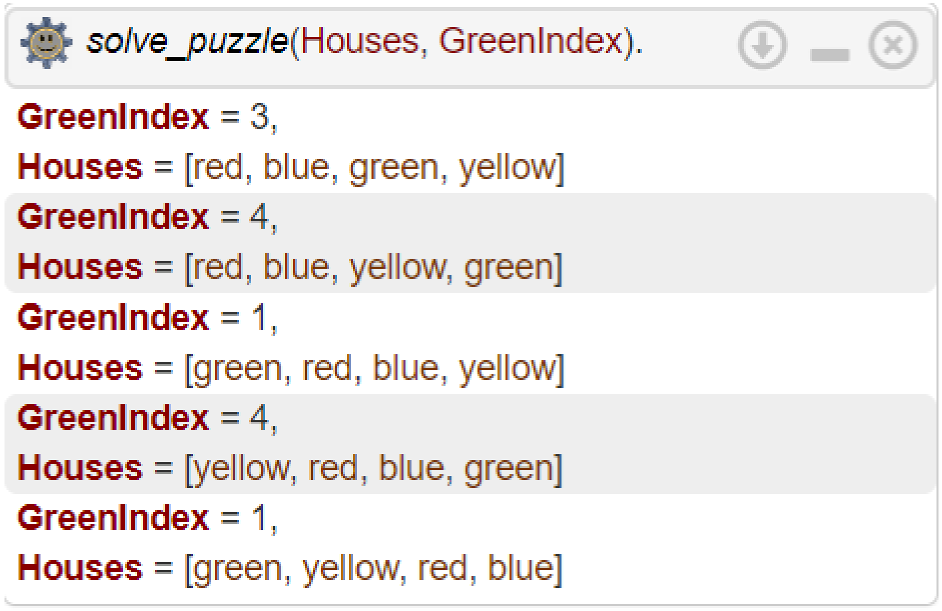
The not_next_to function uses the next_to function, but instead negates the function in order to get the opposite.



```
next_to(X, Y, [X,Y|_]).
next_to(X, Y, [_|T]) :-
 next_to(X, Y, T).

not_next_to(X, Y, List) :
 \+ next_to(X,Y, List).
```

Figure 10. Code Snippet of the next_to and not_next_to functions for Lab 5 Task C.

Step 3. Inputs and Desired Outputs.



```
solve_puzzle(Houses, GreenIndex).

GreenIndex = 3,
Houses = [red, blue, green, yellow]
GreenIndex = 4,
Houses = [red, blue, yellow, green]
GreenIndex = 1,
Houses = [green, red, blue, yellow]
GreenIndex = 4,
Houses = [yellow, red, blue, green]
GreenIndex = 1,
Houses = [green, yellow, red, blue]
```

Figure 11. Inputs and Outputs for the solve_puzzle function for Lab 5 Task C.

The output was not specifically desired, it does show the combinations that are seen as valid solutions.

References.

- ChatGPT (for Prolog grammar and spell-check)
- Previous lab submissions from CSI 2120 for more review on Prolog code and logic.