TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

A LAB REPORT
ON

BACKTRACKING

Lab no.: 4
SUBMISSION DATE: 2080/11/18

SUBMITTED BY:
Name: Darpan Kattel
Roll no.: 077BCT099
Group: B

SUBMITTED TO:
Department of
Electronics and Computer
Engineering

THEORY:

Backtracking

Backtracking is a feature in programming, like in Prolog, where the program tries out different possibilities to solve a goal. It's handy because it saves programmers from having to manually backtrack, but it can sometimes slow down the program.

Imagine you're trying to solve a maze. Backtracking would be like trying one path, hitting a dead end, and then going back to try another path. But sometimes, you don't need to explore every possible path; you just need one solution. That's where backtracking can be a bit inefficient.

To control this, we use something called a 'cut'. It's like saying, "Once you've found a solution, stop looking for more." But using cuts can make the program less clear because it affects the order of operations.

There are two types of cuts:

a. Green cut: When you know some paths won't lead to useful solutions, so you stop looking. This makes the program faster and saves memory.
b. Red cut: When the logic of the program requires you to stop considering certain possibilities.

Here's an example:

Imagine you're looking for a book in a library. You check one shelf, find the book you need, and don't bother checking other shelves. That's like using a cut.

For instance, let's say we have a rule:

```
r1 :- a, b, !, c.
```

This tells the program that once it finds a solution for 'a' and 'b', it should stop looking and move on to 'c'. Even if there are more solutions for 'c', it won't backtrack to 'a' or 'b' or look for another rule for 'r1'.

Example Program:

```
PREDICATES
buy_car(symbol,symbol)
nondeterm car(symbol,symbol,integer)
colors(symbol,symbol)
CLAUSES
buy_car(Model,Color):-
car(Model,Color,Price),
colors(Color,sexy),!,
Price < 25000.
car(maserati,green,25000).
car(corvette,black,24000).
car(corvette,red,26000).
car(porsche,red,24000).
colors(red,sexy).
colors(black,mean).
colors(green,preppy).
GOAL
buy_car(corvette, Y).
```

In this example, the goal is to find a Corvette with a sexy color and a price that's ostensibly affordable. The cut in the buy car rule means that, since there is only one Corvette with a sexy color in the known facts, if its price is too high there's no need to search for another car. Given the goal:

```
buy_car ( corvette , Y )
```

1. Visual Prolog calls car, the first subgoal to the buy car predicate.
2. It makes a test on the first car, the Maserati, which fails.
3. It then tests the next car clauses and finds a match, binding the variable Color with the value black.
4. It proceeds to the next call and tests to see whether the car chosen has a sexy color. Black is not a sexy color in the program, so the test fails.
5. Visual Prolog backtracks to the call to car and once again looks for a Corvette to meet the criteria.
6. It finds a match and again tests the color. This time the color is sexy, and Visual Prolog proceeds to the next subgoal in the rule: the cut. The cut immediately succeeds and effectively ”freezes into place” the variable bindings previously made in this clause.
7. Visual Prolog now proceeds to the next (and final) subgoal in the rule: the comparison Price ¡ 25000.
8. This test fails, and Visual Prolog attempts to backtrack in order to find another car to test. Since the cut prevents backtracking, there is no other way to solve the final subgoal, and the goal terminates in failure.

Structure Revisited

In Prolog, we can use structures to define custom data types according to our needs. For instance, if we want to represent a date, we can define it as a structure in the 'DOMAINS' section like this:

```
date = d(integer, symbol, integer)
```

This means a date consists of three parts: day (integer), month (symbol), and year (integer).

Then, we can use 'date' as a data type to store dates.

```
DOMAINS
date = d(integer, symbol, integer)

PREDICATES
inquire
display(symbol)
date_of_birth(symbol, date)

CLAUSES
date_of_birth(ram, d(12, july, 1983)).
date_of_birth(shyam, d(15, august, 1976)).
date_of_birth(hari, d(26, may, 1994)).
date_of_birth(sita, d(29, september, 1991)).

display(X) :-
    date_of_birth(X, Y),
    write(X), nl,
    write(Y).

inquire :-
    write("Enter the name: "),
    readln(X),
    display(X).
```

The 'inquire' goal prompts the user to enter a name and displays the date of birth associated with that name.

With some adjustments, we can write goals to find people with ages above or below a certain value, or people born in a specific month, just like in a relational database.

In Prolog, facts act as a database. We can think of these facts as similar to tables in a relational database. We can use structures to define relations, and for all integrity purposes, this functions like a table in a relational database. This is often called Prolog's internal database.

We can update this database during program execution using keywords like 'assert' and 'retract':

- **assert(C):** This adds a data into the facts base. asserta(C) and assertz(C) control where the insertion happens; asserta(C) inserts at the beginning, and assertz(C) inserts at the end.
- **retract(C)**: This deletes a clause that matches C from the database.\

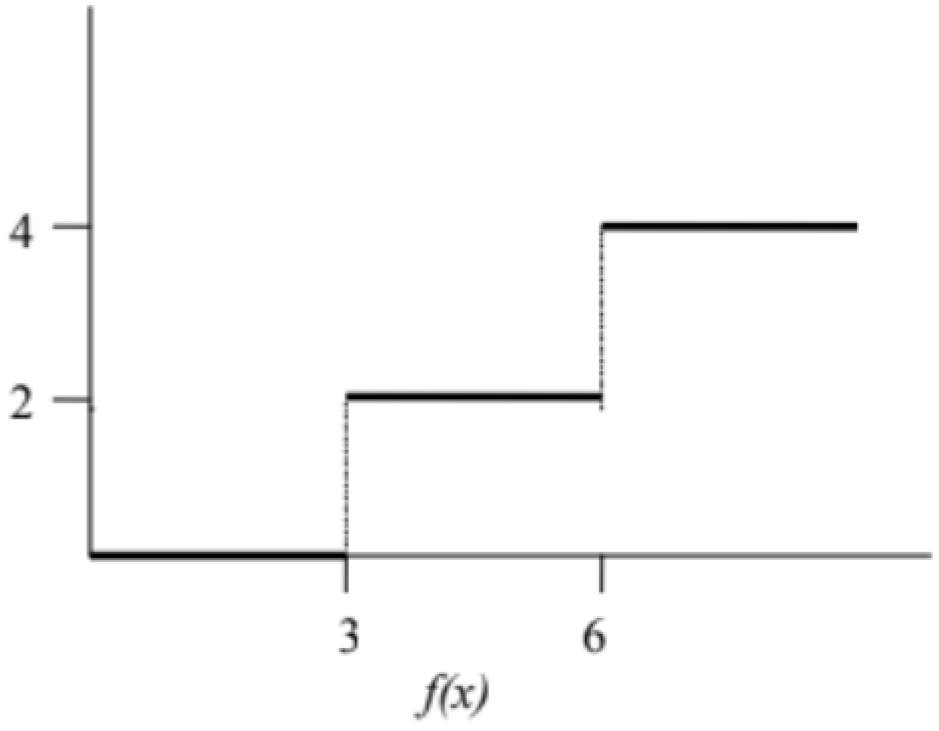
ASSIGNMENTS

Consider the function as shown in the figure below. The relation between X and Y can be specified by the following three rules.

Rule 1: If X < 3, then Y = 0.

Rule 2: If 3 ≤ X < 6, then Y = 2.

Rule 3: If 6 < X, then Y = 4.



A double step function

This can be programmed as:

```
PREDICATES
f ( integer , integer )

CLAUSES
f (X ,0) : - X <3.
f (X ,2) : - 3 <=X ,X <6.
f (X ,4) : - 6 < X .

GOAL
f (2 , X ) .
```

Assignment 1)

Now modify the program using cut and observe the difference between the two modules. Comment on the difference.

```
% Predicates
f (X , 0) : - X < 3 , !.
f (X , 2) : - X >= 3 , X < 6 , !.
f (X , 4) : - X > 6 , !.

% Goal
? - f (2 , X ) .
```

Comment:

In the original version, Prolog tries to satisfy each clause in order, backtracking when necessary. If a clause fails, Prolog backtracks and tries the next clause. This can lead to multiple solutions if there are multiple clauses that match the query.

Modified Version with Cut: In the modified version with the cut operator, once a solution is found, backtracking is prevented. Once a clause successfully matches, Prolog commits to that choice and does not explore alternative solutions for that clause. As a result, if a clause successfully matches, it will not attempt to match subsequent clauses, even if they might also match. This can result in a more deterministic behavior and can prevent unnecessary backtracking.

Assignment 2)

Define the relation min(X,Y,Z) where Z returns the smaller of the two given numbers X and Y. Do it with and without the use of cut and comment on the result.

Solution:

Without cut:

```
min (X , Y , X ) : - X = < Y .
min (X , Y , Y ) : - Y < X .
```

With cut:

```
min (X , Y , X ) : - X = < Y , !.
min (_ , Y , Y ) .
```

In the version without cut, two clauses are defined for the min/3 predicate. The first clause sets Z to X if X is less than or equal to Y. Conversely, the second clause handles cases where Y is strictly less than X, unifying Z with Y. On the other hand, the version with cut employs a single clause with a cut placed after the condition X ≤ Y.

This cut optimizes efficiency by preventing backtracking once a suitable solution is found where X is less than or equal to Y. The second clause serves as a catch-all, using a placeholder for X to match any value, thereby unifying Z with Y in cases where X ≤ Y fails.

```
DOMAINS
date = d ( integer , symbol , integer )
works = w ( symbol , integer )
FACTS
person ( symbol , symbol , date , works ) .
PREDICATES
start
load_name
evalans ( integer )
display
search
dispname ( symbol )
delete
CLAUSES
person ( shyam , sharma , d (12 , august ,1976) ,w ( ntv ,18000)
) .
person ( ram , sharma , d (12 , august ,1976) ,w ( ntv ,18000) ) .
person ( ram , singh , d (13 , may ,2001) ,w ( utl ,12000) ) .
start : -
write (" ************* MENU ***********
*** ") ,nl ,
write (" Press 1 to add new data ") ,nl ,
write (" Press 2 to show existing data ") ,nl ,
write (" Press 3 to search ") ,nl ,
write (" Press 4 to delete ") ,nl ,
write (" Press 0 to exit ") ,nl ,
write (" ************* MENU ***********
*** ") ,nl ,
readint ( X ) ,
evalans ( X ) .
evalans (1) : - load_name , start .
evalans (2) : - display , evalans (2) .
evalans (3) : - search , evalans (3) .
evalans (4) : - delete , evalans (4) .
evalans (0) : - write (" Thank You ") .
delete .
search .
dispname ( N ) : -
```

```
person (N ,C , d (D ,M , Y ) ,w (O , S ) ) ,
write (" Name :",N ," ",C ) ,nl ,
write (" Date of Birth :",D ,"th"," ",M ," ",Y ) ,nl ,
write (" Organisation :",O ) ,nl ,
write (" Salary :",S ) ,nl ,nl.
display : -
retract ( person (N ,X , d (D ,M , Y ) ,w (O , S ) ) ) ,
write (" Name :",N ," ",X ) ,nl ,
write (" Date of Birth :",D ,"th"," ",M ," ",Y ) ,nl ,
write (" Organisation :",O ) ,nl ,
write (" Salary :",S ) ,nl ,nl.
load_name : -
write (" Enter the name \n") ,
readln ( N ) ,
write (" Enter the surname \n") ,
readln ( S ) ,
write (" Date of Birth \n Day:") ,
readint ( D ) ,nl ,
write (" Month :") ,
readln ( M ) ,nl ,
write (" Year :") ,
readint ( Y ) ,nl ,
write (" Enter the organisation :") ,
readln ( O ) ,
write (" Enter the salary :") ,
readint ( Sl ) ,nl ,nl ,
asserta ( person (N ,S , d (D ,M , Y ) ,w (O , Sl ) ) ) .
GOAL
start .
```

Assignment 3)
Observe the above program. Add clauses for search and delete and extend your module as much as you like .

```
delete : -
write (" Enter the name of the person to delete : ") ,
readln ( Name ) ,
retract ( person (Name , _ , _ , _ ) ) ,
write (" Person deleted successfully .") , nl.
search : -
write (" Enter the name of the person to search : ") ,readln ( Name ) ,
( person (Name , Surname , d ( Day , Month , Year ) , w (Organization , Salary )
) ->
write (" Name : ") , write ( Name ) , write (" ") , write (Surname ) , nl ,
write (" Date of Birth : ") , write ( Day ) , write ("th ") ,write ( Month ) ,
write (" ") , write ( Year ) , nl ,
write (" Organization : ") , write ( Organization ) , nl ,
write (" Salary : ") , write ( Salary ) , nl;
write (" Person not found .") , nl
) .
```

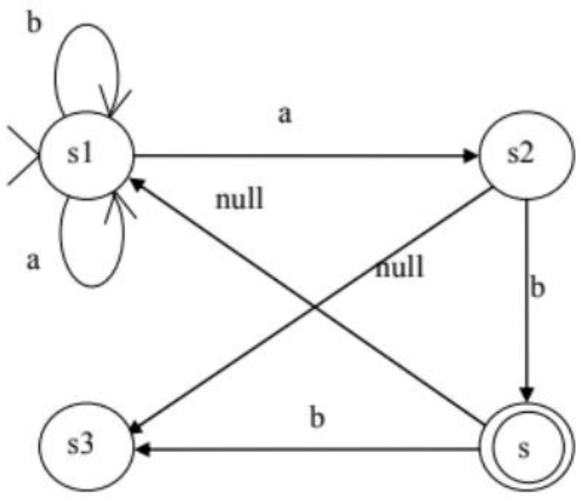
The delete predicate asks the user to input a person’s name for deletion and uses the retract/1 predicate to remove the corresponding fact from the database. Upon successful deletion, it confirms the action with a message.

On the other hand, the search predicate prompts the user for a person’s name to search for. It then tries to find a match within the database’s person facts. If a match is found, it displays comprehensive details including the person’s name, date of birth, organization, and salary. Conversely, if no match is found, it informs the user that the requested person could not be located.

Simple Application

Let us now see how the features of prolog can be used to simulate a nondeterministic automata which would have been a cumbersome task using other programming languages.

A nondeterministic finite automaton is an abstract machine that reads a string of symbols as input and decides whether to accept or to reject the input string. An automaton has a number of states and it is always in one of the states. The automata can change from one state to another upon encountering some input symbols. In a non deterministic automata the transitions can be non deterministic meaning that the transition may take place with a NULL character or the same character may result in different sitions A non deterministic automaton decides which of the possible moves to execute, and it chooses a move that leads to the acceptance of the string if such a move is available. Let us simulate the given automata.



```
DOMAINS
Symb_list = symbol *
PREDICATES
Trans ( symbol , symbol , symbol )
Silent ( symbol , symbol )
Final ( symbol )
CLAUSES
final ( s3 ) .
trans ( s1 ,a , s1 ) .
trans ( s1 ,a , s2 ) .
trans ( s1 ,b , s1 ) .
trans ( s2 ,b , s3 ) .
trans ( s3 ,b , s4 ) .
silent ( s2 , s4 ) .
silent ( s3 , s1 ) .
accepts (S ,[]) : - final ( S ) .
accepts (S ,[ H | T ]) : -
trans (S ,H , S1 ) ,
accepts ( S1 , T ) .
accepts (S , X ) : -
silent (S , S1 ) ,
accepts ( S1 , X ) .
GOAL
Accepts (S ,[ a , b ]) .
```

Output

```
? - goal .
false .
```

Assignment 4)
Check the automaton with various input strings and with various initial states. (The initial state need not necessarily be s1.) Observe the result and comment on how the simulation works. Use the following goals:

1. accepts(s1, [a, a, b])

   ```
   ? - goal .
   true .
   ```

   Accepted by automation starting from state ’s1’, resulting positive outcome.

2. accepts(s1, [a, b, b])

   ```
   ? - goal .
   false .
   ```

   Not accepted by automation starting from state ’s1’, resulting negative outcome.

3. accepts(s, [b, a, b])

   ```
   ? - goal .
   true .
   ```

   Exists at least one state S from which the automaton can accept the sequence [b, a, b], indicating a successful match.

4. accepts(s1, [X, Y, Z])

   ```
   ? - goal .
   false .
   ```

   Acceptance of any sequence represented by variables X, Y, and Z starting from state s1 may not succeed.

5. accepts(s2, [b])

   ```
   ? - goal .
   true .
   ```

   Accepted by the automaton starting from state s2, resulting in a positive outcome.

6. accepts(s1, [_, _, _, _---[a, b]]).

   ```
   ? - goal .
   true .
   ```

   Exists a sequence of at least four symbols followed by the sequence [a, b] that the automaton can accept starting from state s1, leading to a positive outcome.

CONCLUSION

In conclusion, this lab introduced key concepts in Prolog programming, including backtracking and the use of structures to define data types. We explored the application of these concepts through examples such as searching and deleting records in a database. By understanding backtracking, structures, and their practical implementation, we gained insight into building efficient and structured Prolog programs for various problem-solving tasks.