UNIT - IV

How to Prevent backtracking, Examples using cut :

Prolog automatically backtracks when necessary to satisfy a goal. This is a useful feature because it allows the programmer to write declarative code without manually handling alternative choices. However, uncontrolled backtracking can lead to inefficiency, causing unnecessary computations.

Preventing Backtracking with the 'Cut' (!) Operator

To control backtracking, Prolog provides the cut (!) operator. This prevents Prolog from reconsidering previous choices once a certain condition is met.
Example: The Double-Step Function
Consider a function that maps values of X to Y based on specific conditions:
Rules:

1. If $X < 3$, then $Y = 0$.
2. If $3 \leq X < 6$, then $Y = 2$.
3. If $X > 6$, then $Y = 4$.
4.

Without Cut (!): Unnecessary Backtracking

```
double_step(X, 0) :- X < 3.
double_step(X, 2) :- X >= 3, X < 6.
double_step(X, 4) :- X > 6.
```

- This works, but Prolog may still backtrack unnecessarily if a condition fails.
-

With Cut (!): Preventing Backtracking

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

- The ! ensures that once a condition is met, Prolog stops looking for alternative solutions, improving efficiency.
-

When to Use the Cut Operator (!)?

- To eliminate redundant computations in decision-making logic.
- To enforce determinism when multiple rules exist for a relation.
- To optimize performance in large search spaces.

While cut (!) improves efficiency, excessive use can reduce program flexibility. Therefore, it should be used strategically to balance control and declarative clarity.

Explain problems with Negation and cut. Or….
Explain Negation as failure & Examples using cut

"Mary Likes All Animals Except Snakes"
In Prolog, expressing "Mary likes all animals except snakes" requires using negation. A simple way to state that Mary likes all animals is:

```
likes(mary, X) :- animal(X).
```

However, this includes snakes, which must be excluded.

Using fail and cut (!) to Prevent Backtracking
We can express the condition as:

1. If X is a snake, then likes(mary, X) should fail.
2. Otherwise, if X is an animal, Mary likes it.

This is implemented as:

```
likes(mary, X) :-
   snake(X), !, fail.  % If X is a snake, fail immediately
likes(mary, X) :-
   animal(X).  % Otherwise, Mary likes it
```

Here,

- The first rule checks if X is a snake and, if true, the cut (!) prevents Prolog from considering the second rule.
- The fail forces failure, ensuring snakes are never liked.
- If X is not a snake, the second rule applies.

A more compact version combines both rules:

```
likes(mary, X) :- snake(X), !, fail; animal(X).
```

Conclusion

- Negation in Prolog (not/1, \+) helps express "something is not true" logically.
- cut (!) and fail prevent backtracking but must be used carefully to avoid unintended failures.
- Using not/1 makes Prolog rules more readable and closer to natural logic.
- Negation as failure means Prolog assumes something is false if it cannot prove it true, which requires a complete knowledge base to work correctly.

---,

Explain the Problems with cut and negation :

The Use of Cut (!) in Prolog:
The cut (!) facility in Prolog provides benefits but also introduces challenges. Below is a summary of its advantages and disadvantages.
Advantages of Cut (!)

1. Efficiency Improvement
    - Prolog's backtracking mechanism tries multiple solutions, even when they are unnecessary.
    - Cut (!) stops Prolog from considering alternative rules that are guaranteed to fail, improving efficiency.
2. Expressing Mutually Exclusive Rules
    - Example:
    - p :- a, !, b.
    - p :- c.
    - This ensures that if a is true, Prolog does not consider c, making the program more deterministic.

Disadvantages of Cut (!)

1. Loss of Declarative Meaning
    - Without cut, the order of clauses does not affect logic, only performance.

- With cut, changing the order may change results.
- Example:
- p :- a, !, b.
- p :- c.
  vs.
  p :- c.
  p :- a, !, b.
  The second version alters the meaning of p.

2. Risk of Programming Errors
   - Since cut is procedural, misplacing it can lead to incorrect behavior.

likes(mary, X) :- snake(X), !, fail; animal(X).

- This prevents backtracking, changing logical behavior.

Using not/1 Instead of Cut-Fail
Instead of cut-fail, Prolog allows negation as failure:
not(P) :- P, !, fail; true.
Rewriting likes/2 using not/1:
likes(mary, X) :- animal(X), not(snake(X)).

This is more readable and closer to natural logic.

Limitations of Negation (not/1)

1. Negation in Prolog is not Logical Negation
   - not(P) fails if P cannot be proven but does not prove P is false.
   - Example:
   - ?- not(human(mary)).
     - If human(mary) is not in the program, Prolog assumes Mary is not human, which may not be logically correct.
     - This is due to Prolog's "Closed World Assumption"—anything not in the program is assumed false.
2. Order of Execution Matters
   - Example:
   - r(a).
   - q(b).
   - p(X) :- not(r(X)).
     - Query ?- q(X), p(X). returns X = b.
     - Query ?- p(X), q(X). fails due to variable instantiation order.

Conclusions:

- Cut (!) can enhance efficiency but must be used carefully to avoid logic errors.
- Green cuts are safe and can be ignored when reading code.
- Red cuts alter logic and should be used cautiously.
- Negation (not/1) is often preferable to cut-fail but follows Prolog's "Closed World Assumption", leading to potential misunderstandings.
- Despite the risks, cut remains essential in Prolog for efficiency and control.

INPUT & OUTPUT :

Explain Communication with files:

Prolog's standard communication involves user queries and program responses through variable instantiations. While this is simple and practical, it is rigid and lacks flexibility.

Extensions are needed for:

1. Input in Various Forms – e.g., English sentences, not just queries.
2. Custom Output Formats – Formatting output beyond simple Prolog responses.
3. File Handling – Reading from and writing to files instead of only using the user terminal.

File Handling in Prolog

- Prolog allows communication with multiple files through input streams (reading) and output streams (writing).
- The terminal itself acts as a special file, referred to as user.
- Prolog provides two special streams at any time:
    - Current Input Stream (defaults to user)
    - Current Output Stream (defaults to user)

Changing Input and Output Streams

- To read from a file instead of the terminal:
- see(Filename).
- read(Data).
- see(user). % Switch back to terminal
- To write to a file instead of the terminal:
- tell(Filename).
- write(Data).
- tell(user). % Switch back to terminal
- Closing files:
    - seen. → Closes the current input file.
    - told. → Closes the current output file.

File Processing in Prolog

- Sequential Processing Only
    - Files are read linearly (no random access).
    - Once read, data cannot be re-read without reopening the file.
    - Reading at the end of a file returns end-of-file.
- Text Files
    - All files are treated as text files (characters, digits, special symbols).
    - Some characters (like spaces and newlines) are non-printable but affect formatting.

Two Ways to Read and Write Files

1. Character-Based Processing
    - Reads and writes one character at a time.
    - Predicates: get, get0, put.
2. Term-Based Processing
    - Reads and writes entire Prolog terms (structured data).
    - Predicates: read, write.
    - Preferred when data naturally fits Prolog's syntax (e.g., structured facts).

Choosing the Right Method

- Use term-based processing when data follows Prolog syntax (e.g., databases of facts).
- Use character-based processing for more flexible formats (e.g., natural language processing).

Write about Processing files of terms :

The built-in predicate read(X) is used to read a term from the current input stream and unify it with X.

- If X is a variable, it gets instantiated with the read term.
- If read(X) reaches the end of a file, X becomes end_of_file.
- Input terms must end with a full stop (.) followed by a space or newline.
- Failure is final—there is no backtracking to retry input.

The write(X) predicate outputs X in Prolog's standard syntax.

- It can display any term, no matter how complex.
- Additional predicates for formatting output:
  - tab(N): Outputs N spaces.
  - nl: Starts a new line.

Example: Computing Cubes with read and write

A Prolog program to compute cubes of numbers reads user input and prints results:

```
cube :-
   read(X),
   process(X).

process(stop) :- !. % Stop execution when "stop" is entered
process(N) :-
   C is N * N * N,
   write(C), nl,
   cube.
```

Example interaction:

```
?- cube.
2.
8
5.
125
12.
1728
stop.
Yes
```

Each user input must be followed by a full stop.

Common Mistakes

A wrong attempt at simplifying the program:

```
cube :- read(stop), !.
cube :- read(N), C is N*N*N, write(C), cube.
```

Why is this incorrect?

- If a number (e.g., 5.) is read instead of stop., read(stop) fails, and the number is lost.
- If stop. is read by read(N), an error occurs since stop is non-numeric.

Improving User Interaction with Prompts

A better version of the cube program prompts the user before reading:

```
cube :-
   write('Next item, please: '),
   read(X),
   process(X).

process(stop) :- !.
process(N) :-
   C is N * N * N,
   write('Cube of '), write(N), write(' is '), write(C), nl,
   cube.
```

Example interaction:

```
?- cube.
Next item, please: 5.
Cube of 5 is 125
Next item, please: 12.
Cube of 12 is 1728
Next item, please: stop.
yes
```

Depending on the Prolog implementation, an extra request (like ttyflush) may be needed to ensure the prompt appears before reading input.

Explain how the prolog Manipulates characters:

Prolog provides built-in predicates for reading and writing single characters using their ASCII codes.

Writing Characters

The predicate put(C) writes the character corresponding to the ASCII code C to the current output stream.

Example:

```
put(65), put(66), put(67).
```

Output:

```
ABC
(65 = 'A', 66 = 'B', 67 = 'C')
```

Reading Characters

- get0(C) reads a single character and instantiates C to its ASCII code.
- get(C) is a variation that skips non-printable characters (e.g., spaces) before reading the first printable character.

Example:

```
get0(C).
```

If input is 'A', then C = 65.

Example: Removing Extra Spaces from Sentences

The squeeze procedure reads a sentence and reformats it by replacing multiple spaces with a single space.

Example input:

```
The  robot  tried to  pour  wine  out  of the  bottle.
```

Expected output:
The robot tried to pour wine out of the bottle.
Prolog implementation:

```
squeeze :-
   get0(C),
   put(C),
   dorest(C).

dorest(46) :- !.  % 46 is ASCII for a full stop, end processing
dorest(32) :- !,  % 32 is ASCII for a space, skip extra spaces
   get(C),
   put(C),
   dorest(C).

dorest(Letter) :-
   squeeze.
```

How It Works:

1. Reads the first character and prints it.
2. If it's a full stop (ASCII 46) → Stop execution.
3. If it's a space (ASCII 32) → Skip extra spaces before printing the next character.
4. Otherwise, continue reading characters recursively.

This method ensures proper formatting of sentences while eliminating redundant spaces.

What is Constructing and decomposing atoms:

In Prolog, characters can be processed as ASCII codes and converted into atoms using the built-in predicate name/2.
Using name/2 to Convert Atoms and ASCII Codes

- name(A, L) relates an atom A to a list of ASCII codes L.
- Example:
- name('2x2', [50, 120, 50]).
  Here, '2x2' is converted into [50, 120, 50] (ASCII codes of '2', 'x', '2').

Typical Uses of name/2

1. Breaking an atom into its individual characters (useful for pattern matching, prefixes, etc.).
2. Combining a list of characters into an atom (useful for reconstructing words from characters).

Example: Checking If an Atom Represents a Taxi
The program below checks whether an atom starts with "taxi".

```
taxi(X) :-
   name(X, Xlist),
   name(taxi, Tlist),
   conc(Tlist, _, Xlist).  % Check if 'taxi' is a prefix of X
```

Here, conc/3 is a helper predicate for concatenation.

Processing Sentences into Word Lists
To process free-form natural language sentences into lists of atoms, we define:
getsentence(Wordlist).

Example Input:
Mary was pleased to see the robot fail.
Output:
Wordlist = ['Mary', was, pleased, to, see, the, robot, fail].

How the getsentence Procedure Works

1. Reads a character (get0(Char)).
2. Calls getrest/2 to process the character:
    - If Char is a full stop (.) → Sentence ends.
    - If Char is a space ( ) → Skip it and continue reading.
    - If Char is a letter → Read the full word and continue processing.

Prolog Implementation

```
getsentence(Wordlist) :-
   get0(Char),
   getrest(Char, Wordlist).

getrest(46, []) :- !.   % 46 is ASCII for '.'
getrest(32, Wordlist) :- !, getsentence(Wordlist).  % Skip spaces
getrest(Letter, [Word | Wordlist]) :-
   getletters(Letter, Letters, Nextchar),
   name(Word, Letters),
   getrest(Nextchar, Wordlist).

getletters(46, [], 46) :- !.   % End of word at '.'
getletters(32, [], 32) :- !.   % End of word at ' '
getletters(Let, [Let | Letters], Nextchar) :-
   get0(Char),
   getletters(Char, Letters, Nextchar).
```

Usage of getsentence in AI and NLP

- Keyword searching in text.
- Parsing natural language into structured data.
- Understanding sentence meaning, though this is a complex AI task.

This approach is useful for preprocessing text in AI applications before further analysis.

What is consult, reconsult in reading programs?

Prolog allows us to load and manage programs using two built-in predicates: consult and reconsult.

Using consult/1 to Load a Program

- ?- consult(F). loads clauses from file F into Prolog.
- If multiple files are consulted, their clauses accumulate in memory.
- We can also enter programs directly from the terminal with:
- ?- consult(user).
- A shorthand for consulting multiple files:
- ?- [file1, file2, file3].
  This is equivalent to:
  ?- consult(file1), consult(file2), consult(file3).

Using reconsult/1 to Reload a Program

- ?- reconsult(F). behaves like consult/1 but replaces old clauses of previously defined relations instead of just adding new ones.
- Relations not present in F remain unchanged.
- Use reconsult/1 when modifying an existing program instead of simply adding clauses.

Key Difference Between consult/1 and reconsult/1

Predicate	Effect
consult(F)	Adds new clauses to existing ones.
reconsult(F)	Replaces old definitions with new ones from F.

Implementation-Specific Behavior

- The exact behavior of file consulting varies across different Prolog implementations.
- Always check documentation if working with a specific Prolog system.
