Advanced Features of Prolog

The System predicate "cut"

- Prolog is non deterministic in nature because even when a goal has been achieved, the interpreter backtracks to achieve the goal.
- Non deterministic system is one which involves choice points more than one of which lead to a successful conclusion.
- Prolog provides a system defined predicate called *cut* (denoted by !) for affecting the procedural behavior of program and to limit the non determinism by preventing interpreter from finding alternative solutions.

Contd..

- When interpreter comes across a 'cut', the effect is that all alternative solutions of the goal waiting to be tried are abandoned thereby reducing the size of search tree.
- There are many applications, where the very first solution is of interest, if it exists.
- Cut prunes the search tree and hence shortens the path traversed by Prolog interpreter.
- It reduces the computation time and also saves storage space.

Contd…

- We can instruct Prolog interpreter to discard remaining solutions obtained on backtracking by using 'cut'. Semantically 'cut' always succeeds.
- Such controls given to user changes the execution model of Prolog. *Pure Prolog* program is a logic program with specific ordering of clauses in the program and sub goals in the body of a clause.
- These primitives change the normal execution behavior of pure Prolog.

Operational Behavior of Cut

- If a goal G is unified with head of a rule containing 'cut' in its body then,
  - if 'cut' is crossed while satisfying the sub goals in the body, then generation of alternative solutions are debarred using rules with the similar head occurring below that rule i.e., a cut prunes all the rules below it.
  - If a goal G fails after crossing a 'cut', then no alternative rules are tried and goal fails completely.
  - If a goal G fails before crossing a 'cut', then alternative rule, if any with G as head is tried to get solutions.

Contd..

- The cut prunes all alternative solutions to the conjunction of sub goals which appear to the left of 'cut' in the rule body.
- The cut does not affect backtracking amongst the sub goals to the right of cut in the rule body. They can produce more than one solutions.
- If cut is the last sub goal in a rule , then it gives atmost one solution.
- To illustrate cut more clearly, let us consider following rules:

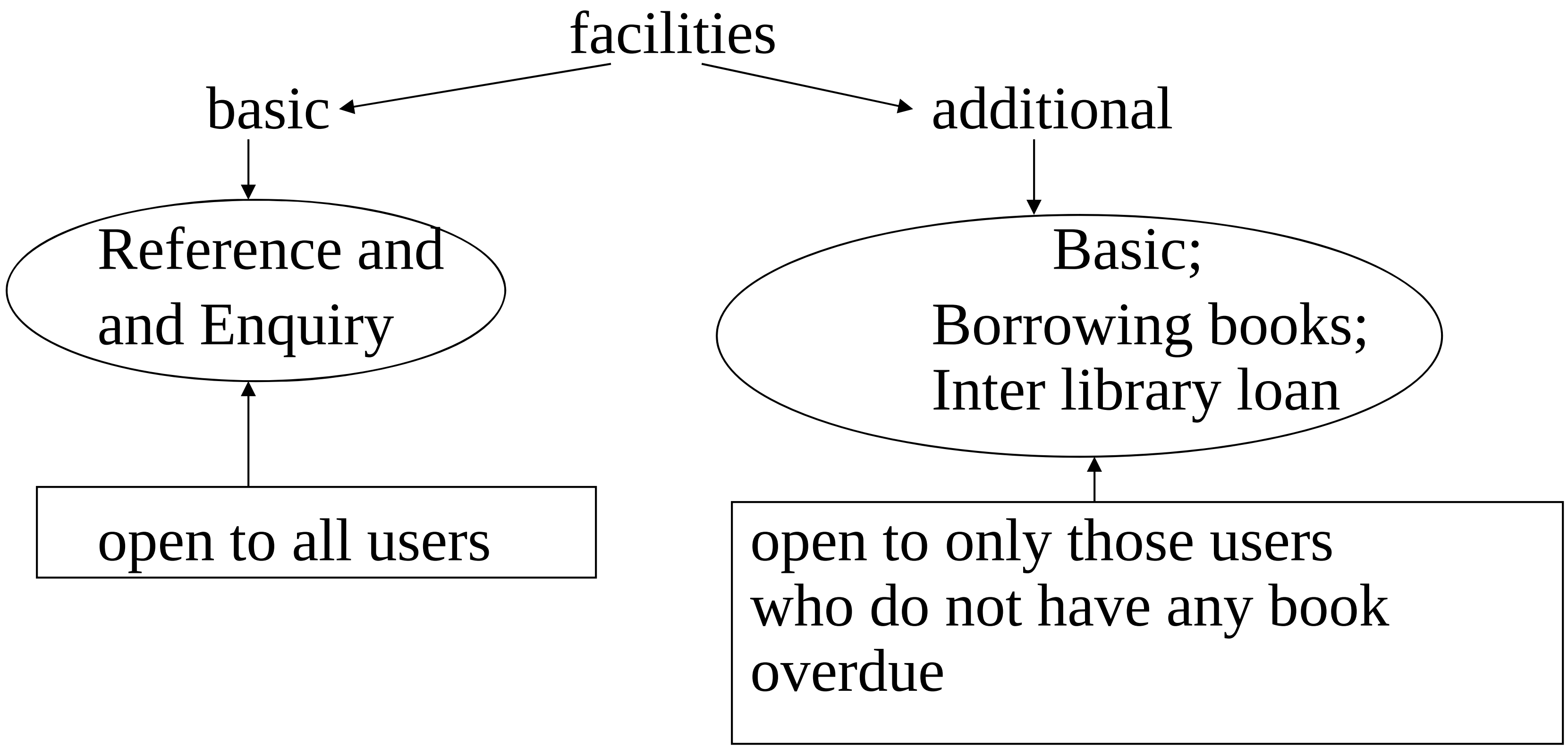
G	:-	A, B, C, !, D, E, F.	(1)
G	:-	P, Q, R.	(2)

Important Uses of Cut

- Program using cuts operates faster because it does not waste time in satisfying those sub goals which will never contribute to the solutions.
- Program may occupy less of memory space because search tree is cut down and does not generate redundant branches.
- If we want to tell a Prolog interpreter that it has found a right rule for a particular *Goal*, then cut is used. Here the cut says "if you get this far, you have picked up the correct rule for this goal”.

Example

Write a program that lists the users and library facilities open to them according to the following scheme.



Prolog Program

```
list (U, F)     :- user (U), facility (U, F).           (1)
facility(U, F):- overdue (U, B), !, basic (F).          (2)
facility  (U,F) :- general (F).                         (3)
basic ( ‘Reference’).
basic (‘Enquiries’).
general (F)   :- basic (F).                             (4)
general (F)   :- additional (F).                        (5)
additional (‘Borrowing’).
additional (‘Inter library loan’).
overdue ('S. K. Das', logic).
user ('S. K. Das').
user ('Rajan').
```

Contd..

- Let us consider another use of cut.
- In Prolog, the rules are of *if-then* type.
- If we are interested in implementing *if-then-else* type of rule in Prolog, then we can make use of cut to do that.
- Define a predicate named as *if_then_else* as follows:

```
/*    if_then_else(U, Q, R) - succeeds  by solving Q if U is
                                  true else by solving R.*/
      if_then_else(U, Q, R) :-        U, !, Q.
      if_then_else(U, Q, R) :-        R.
```

- Operationally it means that "prove U and if succeeds, then prove Q else prove R".
- Declaratively, the relation if_then_else is true if U and Q are both true or if U is not true and R is true.

Types of Cut

- There are two types of cuts viz., green cut and red cut.
- *Green cut* : It does not affect the solution but affects the efficiency of the Prolog program.
- Removal of such cut does not change the meaning of the program.

Example

- Write Prolog program for merging two ordered lists.

```
/*      merge(X, Y, Z)  - Z is obtained by merging
   ordered lists X and Y.  */
merge( [X|X1], [Y|Y1], [X|Z] ) :- X < Y, !,
                                   merge(X1, [Y|Y1], Z).
merge( [X|X1], [Y|Y1], [X, Y|Z] ):- X = Y, !,
                                   merge(X1, Y1, Z).
merge( [X|X1], [Y|Y1], [Y|Z] ) :- X > Y,
                         merge( [X|X1], Y1, Z).
merge(X, [ ], X).
merge( [ ], Y, Y).
```

Red Cut

- The cut whose removal from the program changes the meaning of the program.
- Consider two versions of the programs for finding maximum of two numbers.
- **Version I**

% max(X, Y, Z) – Z is unified with maximum of X and Y.

max(X, Y, Z) :- $X \geq Y$, !, Z = X . (1)

max(X, Y, Y) . (2)

Goals: ?- max(5, 4, 4). Answer: No

?- max (5, 4, 5). Answer: Yes

Contd…

- If the cut is deleted from the rule, then we will not get correct answer.
- **Version 2:**
- If the above program is rewritten as follows, then the cut used here becomes a *green cut* and its removal will not affect the solution (verify).

$$\text{max}(X, Y, Z) \quad :- \quad X > Y, !, Z = X.$$

$$\text{max}(X, Y, Y) \quad :- \quad X \leq Y.$$

Fail predicate

- If we want to force a rule to fail under certain conditions, then built in predicate called *fail* is used in Prolog. Predicate fail tells Prolog interpreter to fail a particular goal and subsequently forces backtracking.
- All the sub goals defined after fail will never be executed.
- Hence predicate fail should always be used as the last sub goal in a rule. It is to be noted that rule containing fail predicate will not produce any solution.

Contd..Example

- Consider the following prolog program

```
listing(Name, Address) :- emp(Name, Address).
emp(ram, cse).
emp(rita, maths).
emp(gita, civil).
```

Goal: ?- listing(Name, Address).

- All possible solutions obtained on executing above goal are:

```
Name = ram  , Address = cse;
Name = rita  , Address = maths;
Name = gita  , Address = civil;
```

Contd..

- The desired results are obtained by normal backtracking of Prolog (finds alternative solutions). Here the variable names are displayed along with the values.
- While developing large software, we might not like to display variable names along with their values but rather values only.
- In order to achieve this, we will change the program as follows and use some more system defined predicates called *write* and *nl* which succeed by writing argument values and creating new line respectively along with fail predicate.

Example using fail predicate

```
listing   :-   write(‘Name‘), write( ‘ Address’), nl,
               emp(Name, Address), write (Name),
               write('     '),write(Address), nl, fail.    (1)
emp(ram, cse).
emp(rita, maths).
emp(gita, civil).
```

Goal: ?- listing

Name	Address
ram	cse
rita	maths
gita	civil

Cut and Fail Combination

- If cut is used in conjunction with fail predicate and if control reaches fail in the body of a rule after crossing cut (!) , then the rule fails and no solution is displayed.
- The reason being that the rules following current rule will not be tried because of cut. The cut-fail combination is a technique that allows early failure.
- A rule with a cut-fail combination says that the search need not proceed.

Example

- Consider the following definitions of the rules using cut and fail combination.

```
X    :-    X1, !, fail.    (1)
X    :-    X2, X3.         (2)
```

Goal: ?- X.

- If X1 succeeds, then rule (1) fails and rule (2) will not be tried.
- If X1 fails, then rule (2) will be tried on backtracking.

Contd..

- Cut and fail combination is also useful for expressing negative facts. For example, "john does not like snakes" could be expressed by the following rule and fact.

```
like(john, snake)    :-        !, fail.
like(john, X).
```

- This coding of rules state that "John likes everything except snake" which implies that "john does not like snakes".

Goal: ?- like(john, snake). Answer: no

Goal: ?- like(john, dog). Answer: yes

Advanced Features in Prolog

Objects in Prolog

- Object is a collection of attributes. Treating related information as a single object is similar to records in conventional programming languages.
- For instance, an object for address can be defined as address(Number, City, Zipcode, Country).
- The entire address is treated as a single object and can be used as an argument of a predicate.
- Here address is called functor and Number, Street, City, Zipcode and Country are called the components which hold actual values.

Contd..

- Consider an example of storing data about a course in two different representations as:
  1. course (logic, monday, 9, 11, saroj, kaushik, block3, room21). (1)
  2. course (logic, time(monday,9,11), lecturer(saroj, kaushik), location(block3, room21)). (2)
- These are two representations of a fact "a lecture course on 'logic', given on Monday from 9 to 11 in the morning by saroj kaushik in block 3, room 21"
- In representation (1), there is a relationship between eight items.

Contd..

- In (2) there is a relationship between four objects – a *course name*, a *time*, a *lecturer* and a *location*.
- The four-argument version of course enables more concise rules to be written by abstracting the details that are relevant to the query.
- Let us define few useful predicates using representation (2) for *course facts*.

teacher_course(L, C) - teacher L teaches a course C.

teacher_on_day(L, Day, C) - teacher L teaches a course C on day Day.

duration(C, D) - course C of D duration.

Contd..

- These predicates are defined as follows:

teacher_course(L, C) :- course(C,_, lecturer(L,_),_).

teacher_on_day(L, Day, C) :-

course(C ,time(Day, _ , _), L , _).

duration(C, D):- course(C, time(-, Start, Finish), _ ,_),

D is Finish – Start.

- Note that we have hidden the details which are not required in particular rule formation by putting underscore (_).
- This is called *Data abstraction*. We don't have definite rules to decide when to use structured data or not.

Queries

Query: Who teaches logic course?

Goal: ?- teacher_course(L, logic).

?- teacher_course(L, logic).

?- course(logic, _, lecturer(L,_), _).

{L = saroj}

succeeds Answer: L = saroj

Query: Which course does saroj teach?

Goal: ?- teacher_course(saroj, C).

?- teacher_course(saroj, C).

?- course(C, _, lecturer(saroj, _), _).

{ C = logic}

succeeds Answer: C = logic

Contd..

Query: Which day does saroj kaushik teach logic course?

? Goal: teacher_on_day(lecturer(saroj, kaushik), X, logic).

?- teacher_on_ day(lecturer(saroj, kaushik), X, logic).

?- course(logic, time(X, _ , _), lecturer(saroj, kaushik), _).

{X = monday}

succeeds

Answer: X = monday

- The representation of time can be changed without affecting those rules which do not inspect time.