Cut & Not

Cut – !

- **Cut**, the name of the **!** operator, is used to
  - **Not waste time on useless choices**
    1. **Know that if current rule fails then trying further rules for the current predicate is useless**
       - **If you got this far then, this is the only rule to try**
    2. **Stop after one solution – do not look for alternate solutions**
    3. **If you continue with this predicate, you will not find a solution**
       - **Use of ! , fail**
- Cut commits to all choices made when entering parent goal – the predicate at the head of the rule
  - **Cannot be re-satisfied on backtracking**

Confirming choice of rule

- Rule 2 for intersection has confirmation use of cut

```
> intersection (A , B , C ) –  A ∩ B = C

intersection ( [] , B , [] ).

intersection ( [ Ah | At ] , B , [ Ah | Ct ] )
   :- member ( Ah , B ) , ! , intersection ( At , B , Ct ).

> Rule 2 is applicable when  head (A ) ∈ B

intersection ( [ Ah | At ] , B , C )
   :- intersection ( At , B , C ).

> Rule 3 is applicable when  head (A ) ∈ B
```

- Once we have established that **Ah** is a member of **B**, then if we backtrack over the member predicate, there is no need to consider rule 3

Stopping – found first solution

- Consider the predicate **sum_to (N ,T)**, where **T** is the sum of the integers 1 .. N

```
sum_to ( 1 , 1 ).
sum_to ( N , T ) :- N1 is N – 1 ,
                    sum_to ( N1 , T1 ) ,
                    T is T1 + N.
```

- The above program works as long as N is an integer >= 1
  - **but there is only one solution, there is no point in trying rule 2 if rule 1 is ever satisfied.**
    - **If ; return is used Prolog loops until memory is exhausted searching for a non existent second solution**

Stopping – found first solution – 2

- So introduce cut into the first case.

```
sum_to ( 1 , 1 )  :-  ! .
sum_to ( N , T )  :-  N1  is  N – 1 ,
                      sum_to ( N1 , T1 ) ,
                      T  is  T1 + N.
```

- Now only one solution is found. Search terminates without infinite loop.
  - **Also example of choice of rule. Once rule 1 has been picked no point in trying rule 2.**

Cut test – 1

- A series of predicates to see how cut works.
- Figure out what the answer will be and then try the queries, **test1(Y)**, **test2(Y)**.

```
w ( X )  :-  nl ,  write ( X ).
w ( X )  :-  write ( ' - w rule 2' ).

test1 ( _ )  :-  w ( 'Path 1' ) , fail ; w ( 'Path 2' ).
test1 ( _ )  :-  w ( 'rule 2 test1' ).

test2 ( _ )  :-  w ( 'Path 1' ) , ! , fail ; w ( 'Path 2' ).
test2 ( _ )  :-  w ( 'rule 2 test2' ).
```

Cut test – 2

- A series of predicates to see how cut works.
- Figure out what the answer will be and then try the queries, **test3(Y), test4(Y)**.

```
w ( X )  :-  nl ,  write ( X ).
w ( X )  :-  write ( ' - w rule 2' ).

test3 ( _ )  :-  ! , w ( 'Path 1' ) , fail ; w ( 'Path 2' ).
test3 ( _ )  :-  w( 'rule 2 test3' ).

test4 ( _ )  :-  ! , ( w ( 'Path 1' ) , fail ; w ( 'Path 2' ) ).
test4 ( _ )  :-  w ( 'rule 2 test4' ).
```

Cut-Fail in action

- **numInRange (X , N)** – 0 ≤ X and X ≤ N
  - > **X = 0 .. n n = any natural number**

    ?- numInRange (X , n).
    X = 0 ; X = 1 ; ... ; X = n ; no

- Definition makes use of cut fail to terminate when n has been reached.
  - > **addUpTo (A ,X , N) means A ≤ X and X ≤ N**
  - > **Acc1 < N is included to have Acc2 <= N, or else infinite search occurs on rule 2 of addUpTo.**

```
numInRange ( X , N )  :-  addUpTo ( 0 , X , N ).
addUpTo ( X , X , N )   :-  X =< N   ;   X > N , ! , fail .
addUpTo ( Acc1 , X , N )
   :- Acc1 < N , Acc2 is Acc1 + 1 , addUpTo( Acc2 , X , N ).
```

Not

- When a rule has the following form

  head :- A , B , C , D.

- You can think of
  - **A as being a guard to trying B, C, D**
  - **A, B as being a guard to trying C, D**
  - **A, B, C as being a guard to trying D**
- For example the use of **member (Ah , B)** in the rule 2 for intersection

Not – 2

- The predicate **not (P)** is used as a guard to select cases as in the following

```
Q ( [ H | T ] , ... )  :-  not ( H = [ _ | _ ] ) , P ( H , ... ) .
  > Only try P if H does not have a head and tail
Q ( [ H | T ] , ... )  :-  not ( H = [ ] ) , P ( H , ... ) .
  > Only try P if H is not the empty list
Q ( [ H | T ] , X , ... )  :-  not ( H = X ) , P ( H , ... ) .
  > Only try P if H is not equal to X
```

Not – Definition

- Not is not built into Prolog as its interpretation depends upon what you want it to mean.

> Prolog searches are based on a **closed universe**
>
> Truth is relative to the database

- Yes means the query can be satisfied by the database
- No means the query cannot be satisfied by the database
  - **It does not mean the query is false, just unsatisfiable**

Not – Definition

- The following is the definition of not as defined in utitlities.pro

```
not ( P )  :-  call ( P ) , ! , fail.

not ( _ ) .
```

- Rule 1 tries **call (P)**
  - **call queries the database with the predicate P – analogous to eval in Lisp**
- If the call succeeds, then the **! , fail** combination says fail and do not try the second rule
  - **So if P gives yes, then not (P) gives no**
- If the call fails, then rule 2 is tried and always succeeds.
  - **So if P gives no, then not (P) gives yes**

Not Definition – Consequence

- The following shows that **not** as defined has side effects
  - **A double negative is not equivalent to a positive!**
- Consider the following expressions

  member (X , [a , b , c]) , write (X).

  > **Finds and writes a, b and c on using ; return**

  not (not (member (X , [a , b , c]))) , write (X).

  > **Succeeds only once with X being a variable**

Not Definition – Consequence – 2

◊ Trace

1 not (not (member (X , [a , b , c])))

2 ==> call (not (member (X , [a , b , c])))

3 ==> call (member (X , [a , b , c]))

succeeds with X = a but ! , fail causes failure

> Failure backs up to 2 , binding to X is lost

> At 2 rule 2 of not is tried and succeeds so call succeeds but ! , fail causes failure

> Failure backs up to 1

> At 1 rule 2 of not is tried and succeeds

> So result is yes with X still a variable

Cut & Not Equivalence

- Cut and not (as defined in slide CN-12) can be used interchangably with a change in rule structure
  - > **Note the use of B as a guard**

```
A :- B , C.
A :- not ( B ) , D.
```

```
A :- B , ! , C.
A :- D.
```

- If **B** succeeds then success or failure of **A** depends upon **C**
- If **B** fails, then success or failure of **A** depends upon **D**

Cut is Dangerous

- Using cut we are taking advantage of the way Prolog searches the database
- Consider the predicate **number_of_parents (X , N)**
  – **X** has **N** parents defined as follows

```
number_of_parents ( adam , 0 )  :-  ! .
number_of_parents ( eve , 0 )  :-  !.
number_of_parents ( X , 2 ).
```

- Definition works correctly if we query such as the following when using **; return** – the cut prevents finding extra solutions for adam and eve

```
number_of_parents ( adam , N ).          ==> 0
number_of_parents ( eve , N ).           ==> 0
number_of_parents ( wilhelma , N ).      ==> 2
```

Cut is Dangerous – 2

- But fails on the following queries

```
number_of_parents ( adam , 2 ).  ==> yes
number_of_parents ( eve , 2 ).   ==> yes
```

- Change the definition to

```
number_of_parents ( adam , N )  :-  ! , N = 0.
number_of_parents ( eve , N )  :-  ! , N = 0.
number_of_parents ( X , 2 ).
```

- Or change the definition to

```
number_of_parents( adam , 0 )  :-  ! .
number_of_parents( eve , 0 )  :-  !.
number_of_parents(X,2)  :-  X \= adam , X \= eve.
```

- Still fail on queries such as the following, expecting backtracking to enumerate all the possibilites

```
number_of_parents ( Who , N ).
```

Cut is Dangerous – Moral

If you introduce cuts to obtain correct behaviour when the goals are of one form, there is no guarantee that anything sensible will happen if goals of another form start appearing.

It follows that it is only possible to use cut reliably if you have a clear policy about how your rules are going to be used. If you change this policy, all the uses of cut must be reviewed.

max (X , Y , M)

- **M** is the maximum of **X** and **Y** if ...

```
max ( X , Y , X ) :- X >= Y , ! .
max ( X , Y , Y ) .
```

- Responds as follows

```
max( 5, 3, 5).      ==> yes
max( 5, 3, Z).      ==> Z = 5
max( 3, 5, 5).      ==> yes
max( 3, 5, Z).      ==> Z = 5
max( 5, 3, 3).      ==> yes   ?????
```

- What happened?
- Lets do a sequence of logical transformations

max (X , Y , M) – 2

```
max ( X , Y, X ) :- X >= Y , ! .
max ( X , Y , Y ) .
```

◊ Is equivalent to ...

```
max ( X , Y , M ) :- M = X , X >= Y , ! .      (1)
max ( X , Y , M ) :- M = Y .
```

◊ Recall the equivalence of **not** and **cut**

```
A :- B , C.                A :- B , ! , C.
A :- not ( B ) , D.        A :- D.
```

◊ Apply to (1)

```
max ( X , Y , M ) :- M = X , X >= Y .
max ( X , Y , M ) :- not ( ( M = X , X >= Y ) ) ,
                     M = Y .
```

max (X , Y , M) – 3

- Have the following equivalence

  not ((P , Q)). not (not (P)) , not (Q).

  not (P).

- Rewrite left column

  not ((P , Q)) ==> ~ (P and Q) ==> ~ P or ~ Q

- Rewrite right column

  not (not (P)) , not (Q) or not (P)

  ==> (~ (~ P) and ~ Q) or ~ P

  ==> (P and ~ Q) or ~ P

  ==> (P or ~ P) and (~ Q or ~ P)

  ==> ~ Q or ~ P

  ==> ~ P or ~ Q

max (X , Y , M) – 4

```
max ( X , Y , M ) :- M = X , X >= Y .
max ( X , Y , M ) :- not ( ( M = X , X >= Y ) ) ,
                     M = Y .
```

- Use the following equivalence in rule 2

```
not( ( P , Q ) ).      not ( not( P ) ) , not( Q ).
                       not( P ).
```

- Gives us

```
max ( X , Y , M ) :- M = X , X >= Y .
max ( X , Y , M ) :- not ( not ( M = X ) ) ,
                     not ( X >= Y ) , M = Y .
max ( X , Y , M ) :- not ( M = X ) , M = Y .
```

max (X , Y , M) – 5

```
max ( X , Y , M ) :- M = X , X >= Y .
max ( X , Y , M ) :- not ( not ( M = X ) ),
                     not ( X >= Y ) , M = Y .
max ( X , Y , M ) :- not ( M = X ) , M = Y .
```

◊ not(not (M = X)) implies M = X, is covered by the remaining terms in rule 1, so it becomes

```
max ( X , Y , M ) :- not ( X >= Y ) , M = Y .
```

> **Giving**

```
max ( X , Y , M ) :- M = X , X >= Y .
max ( X , Y , M ) :- not ( X >= Y ) , M = Y .
max ( X , Y , M ) :- not ( M = X ) , M = Y .
```

max (X , Y , M) – 6

- Rearranging the terms in the first rule gives our final defintion

```
max ( X, Y, M)  :-  X >= Y , M = X.          <==  Correct
max ( X, Y, M)  :-  not( X >= Y ) , M = Y . <== logic
max ( X, Y, M)  :-  not( M = X ) , M = Y .  <== parasitic
```

  > Responsible for max(5, 3, 3) ==> yes

- The logic of the two correct rules guards the equality check **M = X** and **M = Y** , whereas the incorrect solution failed to have a guard on the second rule. We should have

```
max ( X , Y , X )  :-  X  >=  Y  ,  ! .
max ( X , Y , Y )  :-  X  <  Y .
```