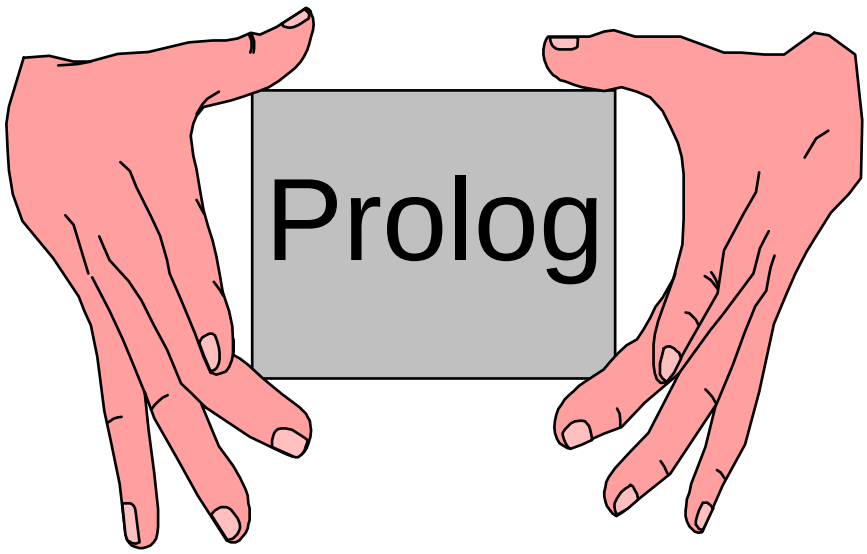


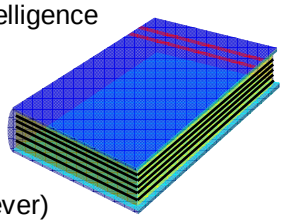
Prolog.1

Textbook and Software

- Title
  - PROLOG programming for artificial intelligence
- Author
  - Ivan Bratko
- Get the software – windows
  - Download PL.zip from the course site
  - Extract the zip file to c:\prolog (or whatever)
  - You should be able to do the same in the pc farm (extract to your personal folder)
  - Run by `c:\prolog\bin\plcon` (or `plwin`)

    Welcome to SWI-Prolog version x.y

    | ?-
- Get the software – csl1
  - Just run `pl`



Prolog.2

Introduction

- What is Prolog?
  - A programming language for symbolic non-numeric computation
- What is programming in Prolog like?
  - Defining relations and querying about relations
- What is SWI-PROLOG?
  - An interpreter
    | ?- main prompt
    | secondary prompt

Prolog.3

The Basics

- Facts:
  `assert(mammal(rat)).`
  `assert(mammal(bear)).`
  `assert(fish(salmon)).`

- Comments:
  /* This is a comment */
  % This is also a comment

- `mammal(bear).`
  yes
- `fish(rat).`
  no
- `fish(X).`
  X = salmon
- `mammal(X).`
  X = rat
  `if we now type a ";" we get the response:`
  X = bear

Prolog.4

More Parameters

```
assert(eats(bear,honey)).
assert(eats(bear,salmon)).
assert(eats(rat, salmon)).
assert(eats(salmon, worm)).
```

Logical AND:

Who eats both honey and salmon:

Which X eats honey and the same X eats salmon?

- `eats(X,salmon) , eats(X,honey).`
  X = bear

Prolog.5

Rules

- For all X and Y
  X is in Y's food chain if
  Y eats X

```
food_chain(X,Y) :- eats(Y,X)
```

The relation foodchain is defined as follows:
`if eats(a,b) then foodchain(b,a)`

- In the prompt, rules should be declared using assert

```
?- assert(food_chain(X,Y) :- eats(Y,X)).
```

- We will usually drop the assert and present only the rule (like it is declared in declaration files)

Prolog.6

Recursive Definitions

For all X and Y
X is in Y's food chain if
Y eats X

or

Y eats some Z and X is in Z's foodchain:

```
food_chain(X,Y) :- eats(Y,X).
food_chain(X,Y) :- eats(Y,Z), food_chain(X,Z).
```

Afterwards we can ask:

```
?-food_chain(X,rat).
X=salmon
;
X=worm
```

Or:

```
?-food_chain(worm,X).
X=rat
;
X=bear
```

Prolog.7

Writing Prolog Programs

- You can use files to declare rules and facts
  - Create a file named "prog" (for example …)
  - Enter the prolog interpreter
  - Type: `consult(prog).`
  - As a result all the facts and rules in the program are loaded.
- In the declarations program you don't need `assert`:
  - Fact: `blue(sea).`
  - Rule: `good_grade(Pupil) :- study(Pupil).`
- At the interpreter prompt you can only type "queries" (or "goals", or "questions")
  - Question: `good_grade(X).`
- To change the database use the goal `assert` (which always succeeds) or retract (which can fail)

```
assert( (good_grade(Pupil) :- study(Pupil)) ).
retract( (good_grade(Pupil) :- study(Pupil)) ).
```

- To give several queries you can use redirection.

Prolog.8

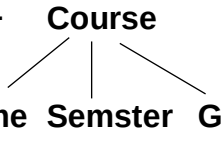
Data Objects

- Both Atoms & Numbers are defined over the following characters:
  - upper-case letters A,B,..,Z
  - lower-case letters a,b,...,z
  - digits 0,1,...,9
  - special characters such as + - * / < > = : . & _ ~
- Atoms can be constructed in 3 ways:
  1. Strings of letters, digits & the underscore, **starting with a lower-case letter.**
     anna x_25 nil
  2. String of special characters
     <----> ::== .:.
  3. Strings of characters enclosed in single quotes:
     'Tom' 'x_>:'
- Reals: 3.14 -0.573
- Integers: 23 5753 -42

Prolog.9

Data Objects

- Variables:
  - strings of letters, digits & "_". **Start with an UPPER-CASE** letter or an "_".
  - `X_25 _result`
- A single "_" is an anonymous variable
  - `getsEaten(X) :- eats(_,X).`
- Facts can have several components, thus looking like Structures.




```
assert(course(pl, fall, 99)).
date(pl, fall, Grade).
Grade= 99
```

Fact Declaration

Query

Prolog.10

Matching

- An operation on terms. Two terms match if:
  - they are identical, or
  - the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.
    - `course(N,S,95)` matches `course(X,fall,G)`
    - `course(N,S,95)` doesn't match `course(Y,M,996)`
    - `course(X)` doesn't match `semester(X)`
- If matching succeeds it always results in the most general instantiation possible.
  - `course(N,M,85) = course(N1,fall,G).`
    N = N1
    M=fall
    G=85

Prolog.11

General rules for matching two terms S and T

(1) If S and T are constants then S and T match only if they are the same object.

(2) If S is a variable and T is anything, then they match, and S is instantiated to T. (or the other way around...)

(3) If S and T are structures then they match only if

(a) S and T have the same principal functor and the same number of components, and

(b) all their corresponding components match.

The resulting instantiation is determined by the matching of the components.

Prolog.12

Geometric Example

- Use structures to represent simple geometric shapes.
  - point - two numbers representing X and Y coordinates.
  - seg - a line defined by two points
  - triangle - defined by three points.
    - `point(1,1)`
    - `seg( point(1,1), point(2,3) )`
    - `triangle( point(4,2), point(6,4), point(7,1) )`
- In the same program we can also use three dimensional points:

  `point(1,3,5)`

  This will result in a different relation with the same name.
- We want to match:

  triangle(point(1,1), A, point(2,3))

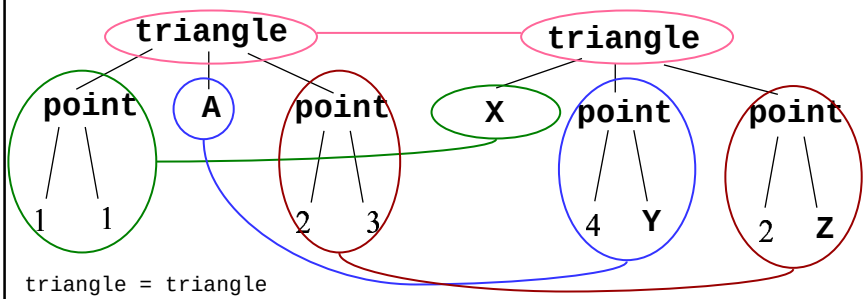
with

  triangle(X, point(4,Y), point(2,Z)).

Prolog.13

Geometric Example

triangle(point(1,1), A, point(2,3)) = triangle(X, point(4,Y), point(2,Z)).




```
triangle = triangle
point(1,1) = X
A = point(4,Y)
point(2,3) = point(2,Z)
```

The resulting instantiation is:

X = point(1,1)

A = point(4,Y)

Z = 3

Prolog.14

Matching as means of Computation

- A program with two facts:
  - `vertical( seg( point(X,Y), point(X, Y1) ) ).`
    `horizontal( seg( point(X,Y), point(X1,Y) ) ).`
- Conversation:
  - ?- `vertical( seg( point(1,1), point(1,2) )).`
    yes
  - ?- `vertical( seg( point(1,1), point(2,Y) )).`
    no
  - ?- `vertical( seg( point(2,3), P)).`
    P = point(2,Y)
- When prolog has to invent a variable name (like the Y above) it will be in the form _n where n is an arbitrary number. So the last line will actually be in Prolog:

  P = point(2,_G501)

Prolog.15

Arithmetics

- Predefined operators for basic arithmetic:
  - +, -, *, /, mod
- If not explicitly requested, the operators are just like any other relation
- Example:
  - `X = 1 + 2.`
    X=1+2
- The predefined operator '*is*' forces evaluation.
  - `X is 1 + 2.`
    X=3
- A **`is`** B (A and B here can be anything) means
  - Evaluate B to a number and perform matching of the result with A
- The comparison operators also force evaluation.
  - `145 * 34 > 100.`
    Yes

Prolog.16

comparison Operators

X > Y X is greater than Y.

X < Y X is less than Y.

X >= Y X is greater than or equal to Y.

X =< Y X is less than or equal to Y.

X =:= Y the values of X and Y are equal.

X =\= Y the values of X and Y are not equal.

Prolog.17

= and =:=

- X = Y causes the matching of X and Y and possibly instantiation of variables.
- X =:= Y causes an arithmetic evaluation of X and Y, and cannot cause any instantiation of variables.

```
1 + 2 =:= 2 + 1.
> yes
1 + 2 = 2 + 1.
> no
1 + A = B + 2.
> A = 2
> B = 1
1 + A =:= B + 2.
> [WARNING: Unbound variable in arithmetic expression]
>  Fail:  (  6) 1 + _G149 =:= _G151 + 2 ?
> No
```

Prolog.18

Example: The Greatest Common Devisor

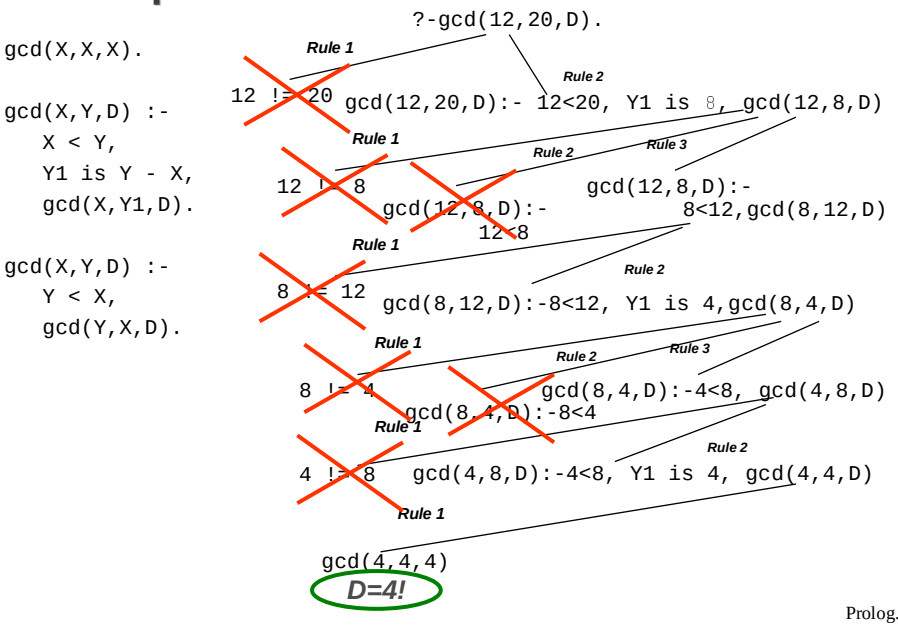
- given X and Y, the gcd D can be found by:
  - (1) If X and Y are equal then D is equal to X.
    (2) If X < Y then D is equal to the gcd of X and (Y-X).
    (3) If Y < X then do the same as in (2) with X and Y interchanged.

```
gcd(X,X,X).
gcd(X,Y,D) :-
   X < Y,
   Y1 is Y - X,
   gcd(X,Y1,D).
gcd(X,Y,D) :-
   Y < X,
   gcd(Y,X,D).
```

Prolog.19

Example: The Greatest Common Devisor

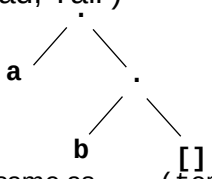
```
gcd(X,X,X).

gcd(X,Y,D) :-
   X < Y,
   Y1 is Y - X,
   gcd(X,Y1,D).

gcd(X,Y,D) :-
   Y < X,
   gcd(Y,X,D).
```



Prolog.20

Lists

- A sequence of any number of items.
- Structure of lists: .(Head, Tail)
  .(a, .(b,[ ])) eq.



- Shorthand:
  - `[tom, jerry]` is the same as `.(tom, .(jerry, []))`
  - `[a | tail]` is the same as `.(a, tail)`
  - `[a,b,c] = [a | [b,c]] = [a,b | [c]] = [a,b,c |[]]`
- Elements can be lists and structures:
  - [a, [1, 2, 3], tom, 1995, date(1,may,1995)]

Prolog.21

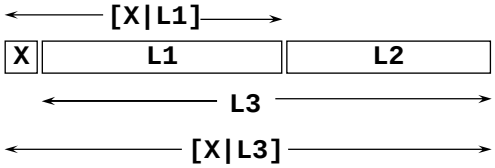
Operations on Lists

- Membership
  - member(X, L) if X is a member of the list L.

```
member(X, [X | Tail]).
member(X, [Head | Tail]) :-
   member(X, Tail).
```

- Concatenation
  - conc(L1, L2, L3) if L3 is the concatenation of L1 and L2.

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :-
   conc(L1, L2, L3).
```



Prolog.22

Examples

```
conc( [a,b,c], [1,2,3], L).
> L = [a,b,c,1,2,3]
conc( L1, L2, [a,b,c] ).
> L1 = []
  L2 = [a,b,c];
> L1 = [a]
  L2 = [b,c];
> L1 = [a,b]
  L2 = [c];
> L1 = [a,b,c]
  L2 = [];
> no
conc( Before, [4|After], [1,2,3,4,5,6,7]).
> Before = [1,2,3]
  After = [5,6,7]
conc(_, [Pred, 4, Succ |_], [1,2,3,4,5,6,7]).
> Pred = 3
  Succ = 5
```

Prolog.23

Examples

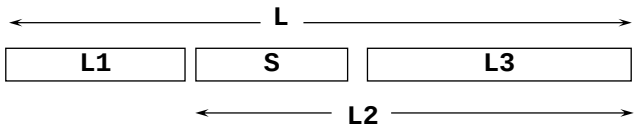
- Redefining member using conc:
  - `member1(X, L) :-`
    `conc(_, [X|_], L).`
- Adding an Item in the front:
  - `add(X, L, [X|L]).`
- Deleting an item
  - `del(X, [X|Tail], Tail).`
    `del(X, [Y|Tail], [Y|Tail1]) :-`
    `del(X, Tail, Tail1).`
  - If there are several occurrences of X in the list then del will be able to delete only one of them.
  - To insert an item at any place in the list:

```
del(a, L, [1,2,3]).
> L = [a,1,2,3];
> L = [1,a,2,3];
> L = [1,2,a,3];
> L = [1,2,3,a];
> no
```

Prolog.24

Examples

- We can define insert using del:
  - ```
    insert(X,List,BiggerList) :-
       del(X, BiggerList, List).
    ```
- The sublist relation
  - ```
    sublist(S, L) :-
       conc(L1, L2, L),
       conc(S, L3, L2).
    ```




  - ```
    sublist(S, [a,b,c]).
    > S = [];
    > S = [a];
    ...
    > S = [b,c];
    ...
    ```

Prolog.25

Permutations

- ```
  permutation([], []).
  permutation([X|L], P) :-
     permutation(L, L1),
     insert(X, L1, P).
  ```
- ```
  permutation( [a,b,c], P).
  > P = [a,b,c];
  > P = [a,c,b];
  > P = [b,a,c];
  ...
  ```
- ```
  permutation2([], []).
  permutation2(L, [X|P]) :-
     del(X, L, L1),
     permutation2(L1, P).
  ```

Prolog.26

Length

- The length of a list can be calculated in the following way:
  - if the list is empty then its length is 0.
  - if the list is not empty then `List = [Head | Tail].` In this case the length is equal to 1 plus the length of the tail `Tail`.
- `length` is built in. If you want to try defining it, change the name...
  -
```
length([], 0).
length([_|Tail],N) :-
   length(Tail, N1),
   N is 1 + N1.
```
  -
```
length([a,b,[c,d],e], N).
> N = 4
length(L,4).
> [_5, _10, _15, _20] ;
.....  ?
```

what happens if the order of these clauses is changed?

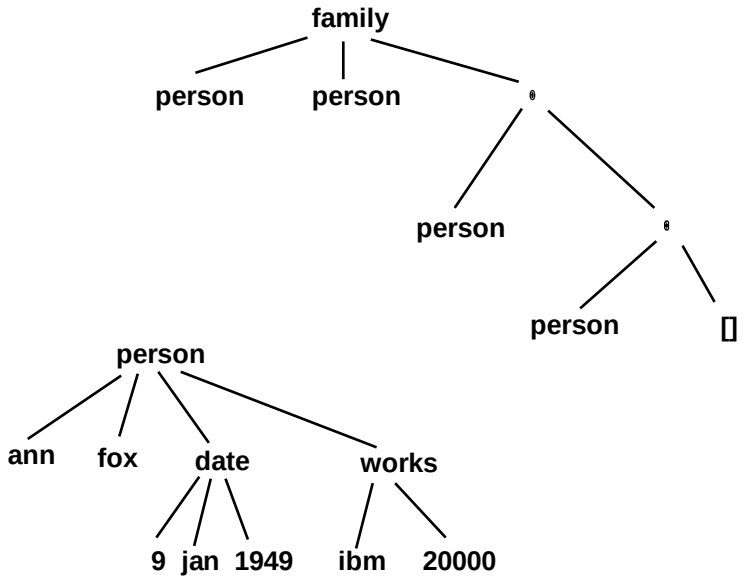
Prolog.27

Database Query

- Represent a database about families as a set of facts. Each family will be a clause.
- The structure of a family:
  - each family has a husband, a wife and children.
  - children are represented as a list.
  - each person has a name, surname, date of birth and job.
- Example:

```
family(
  person(tom, fox, date(7,may,1950), works(bbc,15200),
  person(ann, fox, date(9,jan,1949), works(ibm,20000),
  [ person(pat, fox, date(1,feb,1973), unemployed),
    person(jim, fox, date(4,may,1976), unemployed)]).
```

Prolog.28



Prolog.29

Structure Queries

- All armstrong families:
  - `family( person(_,armstrong,_,_),_,_)`
- Are there families with 3 children?
  - `family(_,_,[_,_,_])`
- Names of families with 3 children.
  - `family(person(_,Name,_,_),_,[_,_,_])`
- All married women that have at least two children:
  - `family(_,person(Name,Surname,_,_),[_,_|_]).`

Prolog.30

Structure Queries

Defining useful relations:

- `husband(X) :- family(X,_,_).`
- `wife(X) :- family(_,X,_).`
- `child(X) :-`
  `family(_,_,Children), member(X, Children).`
  `% the member we`
  `% already defined`
- `exists( Person ) :-`
  `husband(Person); wife(Person); child(Person).`

  Or operator
- `dateofbirth( person(_,_,Date,_),Date).`
- `salary(person(_,_,_,works(_,S)), S).`
  `salary(person(_,_,_,unemployed, 0).`

Prolog.31

Structure Queries

- Names of all people in the database:
  - `exists( person(Name,Surname,_,_)).`
- All employed wives:
  - `wife(person(Name,Surname,_,works(_,_))).`
- Unemployed people born before 1963:
  - `exists(person(Name,Surname,date(_,_,Year),`
    `unemployed)), Year < 1963.`
- People born before 1950 whose salary is less than 8000:
  - `exists(Person),`
    `dateofbirth(Person,date(_,_,Year)),`
    `Year < 1950,`
    `salary(Person, Salary),`
    `Salary < 8000`

Prolog.32

Structure Queries

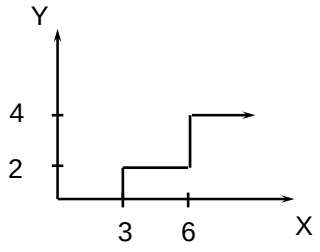
- Calculating the total income of a family:
  - ```
    total([], 0).
    total([Person|List], Sum) :-
      salary( Person, S),
      total(List, Rest),
      Sum is S + Rest.
    ```

    total(People_list , Total_salaries)
  - ```
    tot_income(family(Husband,Wife,Children),I)
    :-
      total([Husband, Wife | Children], I).
    ```
- All families that have an income per family member of less than 2000:
  - ```
    tot_income(family(Husband,Wife,Children), I),
    I/N < 2000.
    ```

Prolog.33

Controlling Backtracking

- Automatic backtracking can cause inefficiency:



- 1. if X < 3 then Y = 0
  2. if 3 <= X and X < 6 then Y = 2
  3. if 6 <= X then Y = 4

Prolog.34
    ```

Controlling Backtracking

- The relation f(X,Y) in prolog would be:
```
f(X,0) :- X<3.
f(X,2) :- 3=<X, X<6.
f(X,4) :- 6=<X.
```
- This procedure assumes that before f(X,Y) is executed X is already instantiated to a number.
- The goal: "`f(1,Y), 2<Y.`" fails, but before prolog replies 'no', it tries all 3 rules.
- The three rules are mutually exclusive so that one of them at most will succeed. If the goal matches the first rule and then fails, there is no point in trying the others.
- The CUT mechanism will help us prevent this.

Prolog.35

Controlling Backtracking: Cut

- A cut prevents backtracking from some point on.
- Written as a '!' sub-goal that always succeeds, but prevents backtracking through it.
- Correcting the example:
```
f(X,0) :- X<3, !.
f(X,2) :- 3=<X, X<6, !.
f(X,4) :- 6=<X.
```
- Whenever the goal f(X,Y) is encountered, only the first rule that matches will be tried.
- If we now ask again "`f(2,Y), 2<Y.`" we will get the same answer, 'no', but only the first rule of 'f' will be tried,
- **note**: the declarative meaning of the procedure did not change.

Prolog.36

Controlling Backtracking: Cut

Another problem:

- If we ask:
  f(7,Y).
  > Y=4
- What happened:
  - 7<3 --> fail
  - 3=<7, 7<6 --> fail
  - 6=<7 --> success.
- Another improvement: The logical rule
  - if X<3 then Y=0,
    otherwise if X<6 then Y=2,
    otherwise Y=4.

  Is translated into:
  - ```
    f(X,0) :- X<3, !.
    f(X,2) :- X<6, !.
    f(X,4).
    ```

Prolog.37

Controlling Backtracking: Cut

- The last change improved efficiency. BUT, removing the cuts now will change the result !!!
  - ```
    f(1,Y).
    > Y = 0;
    > Y = 2;
    > Y = 4;
    >no
    ```
- In this version the cuts do not only effect the procedural meaning of the program, but also change the declarative meaning.

Prolog.38

Controlling Backtracking: Cut

Red and Green cuts:

- When a cut has no effect on the declarative meaning of the program it is called a '**green cut**'. When reading a program, green cuts can simply be ignored.
- Cuts that do effect the declarative meaning are called '**red cuts**'. This type of cuts make programs hard to understand, and they should be used with special care.

Prolog.39

The Meaning of Cut

- When the cut is encountered as a goal it succeeds immediately, but it commits the system to all choices made between the time the parent goal was invoked and the time the cut was encountered.
- H :- B1, B2, ... , Bm, !, ... Bn.

  when the ! is encountered:
  - The solution to B1..Bm is frozen, and all other possible solutions are discarded.
  - The parent goal cannot be matched to any other rule.

Prolog.40

The Meaning of Cut:

- Consider the program
  - C :- P, Q, R, !, S, T, U.
    C :- V.
    A :- B, C, D.

    And the goal: A
  - Backtracking is possible within P,Q,R.
  - When the cut is reached, the current solution of P,Q,R is chosen, and all other solutions are dumped.
  - The alternative clause “C :- V” is also dumped.
  - Backtracking IS possible in S,T,U.
  - The parent goal is “C” so the goal A is not effected. The automatic backtracking in B,C,D is active.

Prolog.41

Examples using CUT

- Adding elements to a list without duplication: add(X,L,L1)
  - If X is a member of L then L1=L.
    Otherwise L1 is equal to L with X inserted:
  - ```
    add(X, L, L) :- member(X, L), !.
    add(X, L, [X|L]).
    ```
- Assume we have the relations ‘big(X)’, ‘medium(X)’ and ‘small(X)’, for example:
  - ```
    big(elephant).
    medium(cat).
    small(mouse).
    ```

  We want a relation ‘boe(X,Y)’ for X is bigger or equal to Y
  - ```
    boe(X,Y) :- small(X),!,small(Y).
    boe(X,Y) :- big(Y),!, big(X).
    boe(X,Y).
    ```

Prolog.42
    ```

Negation

- The special goal fail always fails. (like `1=0.`)
- The special goal true always succeeds. (like `1=1.`)

- "Mary likes all animals but snakes"
  - ```
    likes( mary, X) :- snake(X), !, fail.
    likes( mary, X) :- animal(X).
    ```

- Define the relation "different" by the matching meaning - two terms are different iff they do not match.
  - ```
    different(X, X) :- !, fail.
    different(X, Y).
    ```

Prolog.43

Negation

- Defining "not":
  - if Goal succeeds then not(Goal) fails.
    Otherwise not(Goal) succeeds.
  - ```
    not(P) :- P, !, fail.
    not(P).
    ```
- NOT is a built in prolog procedure, defined as a prefix operator:
  - `not(snake(X)) ==> not snake(X)`
- Previous examples that use the combination "!, fail" can now be rewritten:
  - `different(X, Y) :- not (X = Y).`

Prolog.44
    ```

The 'Not' Operator:

- When possible, it is better to use 'not' than to use the 'cut and fail' combination.
- Note that if the goal "not(A)" succeeds it does not mean that "A is not true" but that "given the current database, A cannot be proved".
- Can you explain the following results:
  - ```
    assert(r(a)).
    assert(q(b)).
    assert(p(X) :- not r(X)).
    ```
  - ```
    q(X), p(X).
    > X = b.
    ```
  - ```
    p(X), q(X).
    > no
    ```

Prolog.45
    ```