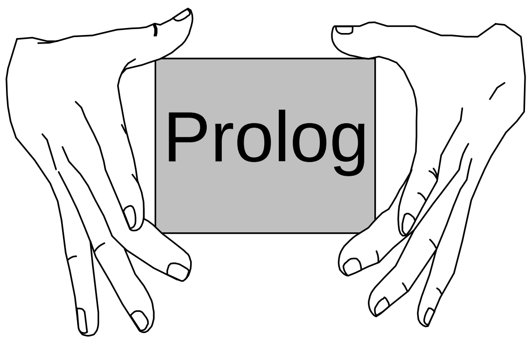


Prolog.1

Textbook

- Title
  - PROLOG programming for artificial intelligence
- Author
  - Ivan Bratko
- We will be using CPROLOG
  - Make sure that `/usr/local/bin` is in your path.
- To enter type "`cprolog`" at your T2/TX shell prompt.
  - `cprolog`

    ```
    C-Prolog version x.y
    | ?-
    ```

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 CPROLOG?
  - An interpreter

    ```
    | ?-     main prompt
    |        secondary prompt
    ```

Prolog.3

Family Tree

Facts:

```
assert(parent(pam,bob)).
assert(parent(tom,bob)).
assert(parent(tom,liz)).
assert(parent(bob,ann)).
assert(parent(bob,pat)).
assert(parent(pat,jim)).
```

- `parent(bob,pat).`
  yes
- `parent(liz,pat).`
  no
- `parent(X,liz).`
  X = tom
- `parent(bob, X).`
  X = ann
  `if we now type a “;” we get the response:`
  X = pat

Who is a parent of whom?

- Find X and Y such that X is a parent of Y
  - `parent(X,Y).`
    X = pam
    Y = bob ;
    X = tom
    Y = bob ;
    ...
- Logical AND:
  Who is a parent, X, of Ann?
  Is (this same) X a parent of Pat?
  - `parent(X,ann) , parent(X, pat).`
    X = bob

Rules

- For all X and Y
  Y is an offspring of X if
  X is a parent of Y
  - `offspring(Y,X) :- parent(X,Y).`
  - The relation offspring is defined as follows:
    `if parent(a,b) then offspring(b,a)`
  - parent and offspring are binary relations.
    The relation parent was defined by explicitly naming it’s couples. The relation offspring is defined by the above rule.

Recursive Definitions

- For all X and Z
  X is a predecessor of Z if
  X is a parent of Z.
- For all X and Z,
  X is a predecessor of Z if
  there is a Y such that
  (1) X is a parent of Y and
  (2) Y is a predecessor of Z.

  -
    ```
    predecessor(X,Z) :-
      parent(X,Z).
    predecessor(X,Z) :-
      parent(X,Y),
      predecessor(Y,Z).
    ```

  -
    ```
    predecessor(pam,X).
    X = bob ;
    X = ann ;
    X = pat ....
    ```

Comments

- /* this is a comment */
- % This is also a comment

Syntax and Meaning

- Simple data objects
- Structured objects
- Matching as the fundamental operation on objects
- Declarative (or non-procedural) meaning of a program
- Procedural meaning of a program
- Relation between the declarative and procedural meanings of a program
- Altering the procedural meaning by reordering clauses and goals

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.

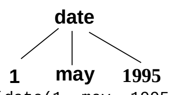
     anna x_25 nil
  2. string of special characters

     <----> ::== .:.
  3. strings of characters enclosed in single quotes:

     'Tom' 'x_>:'
- Reals: 3.14 -0.573
- Integers: 23 5753 -42

Prolog.10

- Variables:
  - strings of letters, digits & "_". Start with an UPPER-CASE letter or an "_".
  - `X_25    _result`
- A single "_" is an anonymous variable
  - `haschild(X) :- parent(X,_).`
- Structures: objects that have several components.



```
assert(date(1, may, 1995)).
date(Day, may, 1995).
```
Day = 1

Prolog.11

Writing Prolog Programs

- How to load a program into the prolog interpreter
  - create the file named "prog".
  - enter the cprolog interpreter.
  - type: consult(prog).
  - as a result all the facts and rules in the program are loaded.
- Prolog consists of clauses. There are 3 types of clauses.
- Only facts and rules clauses can appear in a program
  - facts: `blue(sea).`
  - rules: `good_grade(Pupil)  :-  study(Pupil).`
- At the interpreter prompt you can only type goals
  - questions: `good_grade(X).`
- To change the database use the goal *assert.* (Which always succeeds.)

```
assert( blue(sea) ).
```

Prolog.12

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.
    - `date(D,M,1995)` matches `date(D1,may,Y1)`
    - `date(D,M,1995)` doesn't match `date(D1,M1,1996)`
    - `date(D)` doesn't match `day(D)`
- If matching succeeds it always results in the most general instantiation possible.
  - `date(D,M,1995) = date(D1,may,Y1).`
    D = D1
    M=may
    Y1=1995

Prolog.13

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.

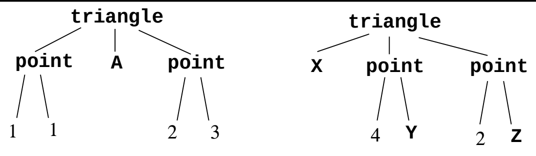
The resulting instantiation is determined by the matching of the components.

Prolog.14

An Illustration

- 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.
- Match: triangle(point(1,1), A, point(2,3)) &
  triangle(X, point(4,Y),point(2,Z))

Prolog.15



match by:

```
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.16

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 cprolog has to invent a variable name (like the Y above) it will be in the form _n where n is an arbitrary number.

Prolog.17

Declarative vs. Procedural

- Given the rule   P :- Q,R.
  - P is true if Q and R are true.
  - From Q and R follows P.
  - To solve problem P, first solve the subproblem Q, and then the subproblem R.
  - To satisfy P, first satisfy Q and then R.
- First two interpretations are called the *declarative meaning*.
  The declarative meaning is concerned only with the relations defined by the program.
- Last two interpretations are called the *procedural meaning*.
  The procedural meaning determines how the relations are actually evaluated by the system.
- P is called the *head* and Q,R is called the *body*.

Prolog.18

Instance of a Clause

- Let C be a clause
  - hasachild(X) :- parent(X,Y).
- An instance of a clause C is the clause C with each of its variables substituted by some term.
  - hasachild(peter) :- parent(peter,Z).
  - hasachild(barry) :- parent(barry,small(caroline)).
- A variant of a clause C is such an instance of the clause C where each variable is substituted by another variable.
  - hasachild(A) :- parent(A,B).
  - hasachild(X1) :- parent(X1,X2).

Prolog.19

Satisfiable

- A goal is true if and only if
  - there is a clause C in the program such that
  - there is a clause instance I of C such that
    - the head of I is identical to G, and
    - all the goals in the body of I are true.
- A comma between goals denotes a conjunction of goals. A semicolon between goals denotes disjunction of goals.
- `P :- Q;R.`
  - means “P is true if Q is true or R is true”.
  - This is the same as:

```
P :- Q.
P :- R.
```

Prolog.20

Satisfaction of Goals

- When prolog is given a goal, or a sequence of goals, it tries to *satisfy* them. If it succeeds it answers yes and specifies how it was done. If not, it answers no.
- Facts and rules are accepted as axioms. The question is considered a theorem to be proved.
- Prolog starts with the given goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts.
- Prolog uses a BACKTRACKING mechanism to generate all possible instantiations while trying to satisfy a goal. To backtrack means that if an instantiation fails, prolog goes back to the last place where a fact or rule was chosen, and looks for the next matching fact or rule that can be used. If there is no such fact or rule, prolog backtracks another step, and so on.

Prolog.21

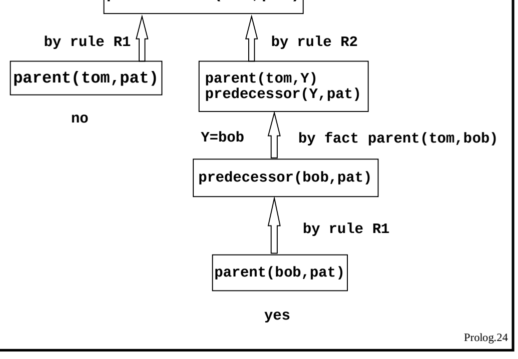
```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

offspring(Y,X) :-
  parent(X, Y).

predecessor(X,Z) :-      % rule R1
  parent(X, Z).
predecessor(X,Z) :-      % rule R2
  parent(X,Y),
  predecessor(Y,Z).
```

Prolog.22

- Example: given the goal
  `predecessor( tom, pat).`
  - first rule found that can match is R1. =>
    X = tom, Z= pat
  - `parent(tom, pat)` => fail
  - backtrack one step to the goal `predecessor(tom, pat).`
    next rule that matches is R2 =>
    `X = tom, Z = pat.` (Y not yet instantiated)
  - `parent(tom, Y), predecessor(Y, pat)`
    prolog now tries to satisfy these goals in the order they appear.
  - `parent(tom, Y)` matches the fact `parent(tom, bob)`
    resulting in `Y = bob`.
  - `predecessor(bob, pat)`
    first rule that matches is R1. =>
    `X' = bob, Z' = pat`
  - `parent(bob, pat) => yes`

Prolog.23

Graph of Execution



Prolog.24

Operator Notation

- Defining a relation as an operator:
  - `:- op( precedence,  type,  name)`
  - This is a special kind of clause called a *directive*.
  - `name` is an atom.
  - `precedence` is an integer. The range is implementation dependent. In your programs you should use the range 1-999. The operator with lowest precedence binds the strongest.
  - There are three groups of types:
    - xfx xfy yfx - infix operators
    - fx fy - prefix operators
    - xf yf - postfix operators
  - f represents the operator, x & y represent arguments
- Example:
  - `:- op(600, xfx, wants).`   (in program or at prompt)
  - `tim wants water.`

Prolog.25

Precedence of Arguments

- The precedence of an atom is 0. The precedence of a structure is that of its principal functor.
- X represents an argument whose precedence must be strictly lower than that of the operator.
- Y represents an argument whose precedence must be lower or equal to that of the operator.
- Examples:
  - a - b - c   Is normally understood as (a - b) - c. =>
    the operator '-' has to be defined as y f x.
  - If the operator not is defined as fy then the expression
    `not not p`   is legal. If it is defined as fx it is illegal, and will have to be written as: `not (not p)`.
- The readability of programs can be often improved by using the operator notation.
- In principle, no operation on data is associated with an operator except in special cases.

Prolog.26

Predefined Operators in Prolog

```
:- op(1200, xfx, ':-').
:- op(1200, fx, [:-, ?-]).
:- op(1100, xfy, ';').
:- op(1000, xfy, ',').
:- op(700, xfx, [=,is,<,>,=<,>=,==,=\=,\==,=:=]).
:- op(500, yfx, [+,-]).
:- op(500, fx, [+, -, not]).
:- op(400, yfx, [*, /, div]).
:- op(300, xfx, mod).
```

Prolog.27

Arithmetics

- Predefined operators for basic arithmetic:
  - `+, -, *, /, mod`
- This is a special case in which an operator may invoke an operation. It will be done only if it is explicitly indicated.
- Example:

  -
    ```
    X = 1 + 2.
    X=1+2
    ```
- The predefined operator '*is*' forces evaluation.

  -
    ```
    X is 1 + 2.
    X=3
    ```
- Operators that were defined as yfx associate to the left. Operators that were defined as xfy associate to the right.
- The comparison operators also force evaluation.

  -
    ```
    145 * 34 > 100.
    yes
    ```

Prolog.28

The 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.29

= 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
```

Prolog.30

Example: The Greatest Common Devisor

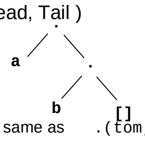
- given X and Y, the gcd D can be found by:
  - (1) If X and Y are equal then D is equal to X.
    (2) If X < Y then D is equal to the gcd of X and (X-Y).
    (3) If Y < X then do the same as in (2) with X and Y interchanged.

```
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.31

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.32

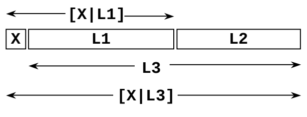
Operations on Lists

- 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.33

```
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.34

- Redefining member using conc:

  -
    ```
    member1(X, L) :-
        conc(_, [X|_], L).
    ```
- Adding an Item in the front:

  -
    ```
    add(X, L, [X|L]).
    ```
- Deleting an item

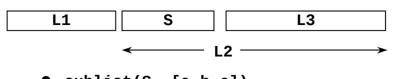
  -
    ```
    del(X, [X|Tail], Tail).
    del(X, [Y|Tail], [Y|Tail1]) :-
        del(X, Tail, Tail1).
    ```
  - If there are several occurrences of X in the list then del will be able to delete anyone of them.
  - To insert an item at any place in the list:
    ```
    del(a, L, [1,2,3]).
    > L = [a,1,2,3];
    > L = [1,a,2,3];
    > L = [1,2,a,3];
    > L = [1,2,3,a];
    > no
    ```

Prolog.35

- 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.36

- 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).
    ```

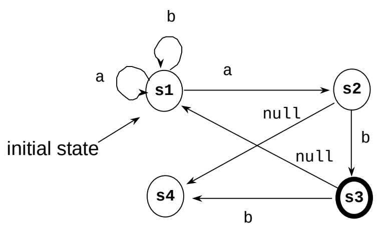
Length

- 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.
    ```
    what happens if the order of these clauses is changed?

  -
    ```
    length([a,b,[c,d],e], N).
    > N = 4
    length(L,4).
    > [_5, _10, _15, _20] ;
    .....  ?
    ```

A Non-deterministic Automata



The language: L = {a,b}*ab

- The automata accepts a given string if there is a transition path in the automata graph such that:
  - it starts with the initial state,
  - it ends with a final state, and
  - the arc labels along the path correspond to the string.
- We can specify an automaton by three relations:
  - final: a unary relation which defines the final states.
  - trans: a three-argument relation which defines the state transitions.
    - trans(S1,X,S2) iff a transition from S1 to S2 is possible when the current input symbol is X.
  - silent: a binary relation defining the silent moves.
    - silent(S1,S2) iff a silent move is possible from S1 to S2.

- A program defining the automaton from slide 34:
  - `final(s3).`

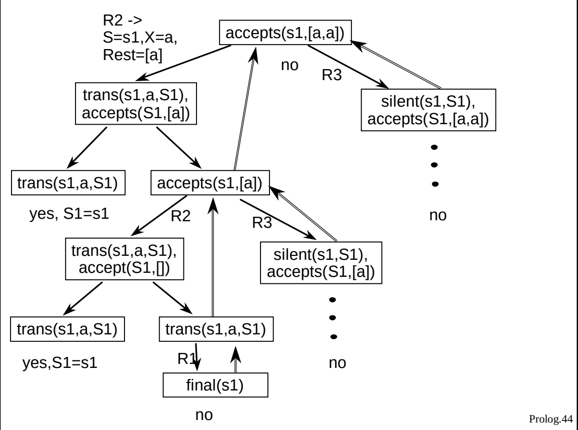
```
trans( s1, a, s1).
trans( s1, a, s2).
trans( s1, b, s1).
trans( s2, b, s3).
trans( s3, b, s4).

silent( s2, s4).
silent( s3, s1).
```

- Input strings will be represented as lists. The string `aab` will be the list `[a,a,b]`.
- The simulation of the automaton will be a binary relation `accepts` which defines the acceptance of a string from a given state.
  - `accepts(State, String)` iff starting from State, the automaton accepts String.

- Logical definition of accepts:
  - [] is accepted from a state S if S is final.
  - A non-empty string is accepted from S if reading the first symbol in the string can bring the automaton into some state S1, and the rest of the string is accepted from S1.
  - A string is accepted from a state S if the automaton can make a silent move from S to S1 and then accept the string from S1.
- In prolog:

```
accepts( S, []) :-          % R1: empty string
   final( S).
accepts( S, [X|Rest]) :-    % R2: read a symbol
   trans( S, X, S1),
   accepts( S1, Rest).
accepts( S, String) :-      % R3: silent move
   silent( S, S1),
   accepts( S1, String).
accepts( s1, [a,a,a,b]).
> yes
accepts(S, [a,b]).
> S = s1;
> S = s3
accepts(s1, [X1,X2,X3]).
> X1 = a
> X2 = a
  X3 = b;
> X1 = b
  X2 = a
  X3 = b;
> no
String = [_,_,_], accepts( s1, String).
> String = [a,a,b];
> String = [b,a,b];
> no
```

Prolog.43



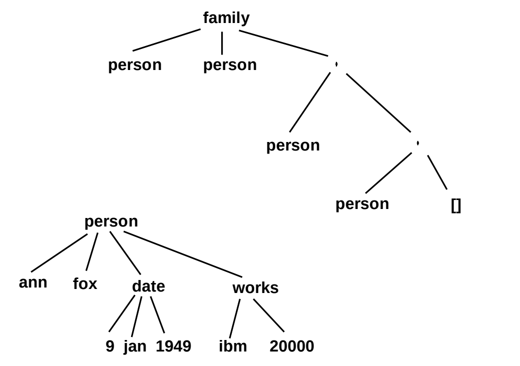
Prolog.44

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.45



Prolog.46

Searching with Structures

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

Prolog.47

- Some useful utility procedures:

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

Prolog.48

Using the utilities:

- Names of all people in the database:
  - `exists( person(Name,Surname,_,_)).`
- All children born in 1981:

  -
    ```
    child(X),
    dateofbirth(X, date(_,_,1981)).
    ```
- 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.49

- To calculate the total income of a family:
  total(List_of_people, Som_of_their_salaries)

  -
    ```
    total([], 0).
    total([Person|List], Sum) :-
      salary( Person, S),
      total(List, Rest),
      Sum is S + Rest.
    ```

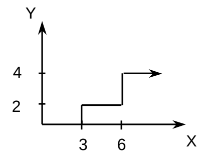
  -
    ```
    tot_income(family(Husband,Wife,Children),I)
    :-
      total([Husband, Wife | Children], I).
    ```
- All families that have an income per family member of less than 2000:

  -
    ```
    tot_income(family(Husband,Wife,Children), I),
    I/N < 2000.
    ```

Prolog.50

Controlling Backtracking

- Automatic backtracking can cause inefficiency:



o 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.51

- 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.52

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.53

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.54

- The last change improved efficiency. BUT, removing the cuts now will result in multiple answers, some of which are not correct.

  -
    ```
    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.55

The meaning of the CUT:

- When matching a goal G to a rule H :- Body, G is called the "parent goal".
- 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.56

- 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.57

Examples using CUT

- The maximum of two elements:
  - `max(X,Y,Max) <=> Max is the maximum of X and Y.`

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

  A more economical solution (using “otherwise” logic):

  -
    ```
    max(X, Y, X) :- X >= Y, !.
    max(X, Y, Y).
    ```
- A single-solution membership:

  -
    ```
    member(X, [X | L]) :- !.
    member(X, [Y | L}) :- member(X, L).
    ```

Prolog.58

- 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]).
    ```
- Classification into categories.

  -
    ```
    beat(tom, jim).
    beat(ann, tom).
    beat(pat, jim).
    ```
  - <u>winner</u>: every player who won all his or her games.
    <u>fighter</u>: any player that won some games and lost some.
    <u>sportsman</u>: any player who lost all his or her games.

  -
    ```
    class(X, fighter) :-
       beat(X, _), beat(_,X), !.
    class(X, winner) :-
       beat(X, _), !.     <-------    cut not necessary
    class(X, sportsman) :-
       beat(_,X), !.
    ```

Prolog.59

Negation

- The special goal fail always fails.
- The special goal true always succeeds.
- “Mary likes all animals but snakes”

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

  -
    ```
    different(X, Y) :-
      X = Y, !, fail;
      true.
    ```

Prolog.60

- Defining “not”:
  - if Goal succeeds then not(Goal) fails.
    Otherwise not(Goal) succeeds.

  -
    ```
    not(P) :-
       P, !, fail;
       true.
    ```
- 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.61

Problems with ‘`cut`’ and ‘`fail`’

- Using CUT can lose the correspondence between the declarative meaning and the procedural meaning.
- When there are no cuts in a program, changing the order of clauses will not change the meaning.
- But:

```
p :- a, b.
p :- c.                 means p <==> (a & b) v c

p :- a, !, b.
p :- c.                 means p <==> (a & b) v (~a & c)

p :- c.
p :- a, !, b.           means p <==> c v (a & b)
```

Prolog.62

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.

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”.

Prolog.63

Can you explain the following results:

-
  ```
  r(a).
  q(b).
  p(X) :- not r(X).
  ```

-
  ```
  q(X), p(X).
  > X = b.
  ```

-
  ```
  p(X), q(X).
  > no
  ```

Prolog.64

Built-in Procedures

Testing the type of terms:

- Types of terms may be variable, integer, atom etc.. A term of type variable may be instantiated or not. If it is, its value can be of type atom, structure et.
- Built-in predicates:

integer(X)	<=>	X is an integer
var(X)	<=>	X is an uninstantiated variable
nonvar(X)	<=>	X is a term other than a variable, or an already instantiated variable
atom(X)	<=>	X currently stands for an atom
atomic(X)	<=>	X currently stands for an integer or an atom.

Prolog.65

- Examples:

  -
    ```
    var(Z), Z=2.
    > Z=2
    Z=2, var(Z).
    > no
    ```

  -
    ```
    integer(Z), Z=2.
    > no
    var(Z), Z=2, integer(Z), nonvar(Z).
    > Z=2
    ```

  -
    ```
    atom(22).
    > no
    atomic(22).
    > yes
    atom(==>).
    >yes
    atom( date(1, may, 1995) ).
    > no
    ```

Prolog.66

- Using these predicates:
  - `...(integer(X), integer(Y), Z is X+Y;`
    `     what-to-do-incase-of-failure) ...`
  - count(A,L,N) <=> A appears N times in the list L

```
count(_,[],0).
count(A, [A|L], N) :- !,
  count(A, L, N1),
  N is N1 + 1.
count(A, [_|L],N) :-
  count(A, L, N).
```

  - but then:

```
count(a, [a,b,X,Y], N).
> N = 3
 count(b, [a,b,X,Y], N).
> N = 3
```

  - X and Y were instantiated to a (to b).

- new solution:

```
count(_, [], 0).
count(A, [B|L], N) :-
  atom(B), A = B, !,    %B is atom A?
    count(A, L, N1),    %count in tail
    N is N1 + 1;
  count(A, L, N).       %otherwise - count in tail
```

Kinds of Equality

- X = Y is true if X and Y match.
- X is E is true if X matches the value of the arithmetic expression E.
- E1 =:= E2 is true if the values of the arithmetic expressions E1 and E2 are equal.
- E1 =\= E2 is true if the values of the arithmetic expressions E1 and E2 are not equal.
- T1 == T2 is true if the terms T1 and T2 are *literally* the same. That is they have exactly the same structure and all the corresponding components are the same (including variable names).
- T1 \== T2 is true if T1 == T2 is not.

- Examples:

  -
    ```
    f(a,b) == f(a,b).
    > yes
    ```

  -
    ```
    f(a,b) == f(a,X).
    > no
    ```

  -
    ```
    f(a,X) == f(a,Y).
    > no
    ```

  -
    ```
    X \== Y
    > yes
    ```

  -
    ```
    t(X, f(a,Y)) == t(X, f(a,Y)).
    > yes
    ```

Database Manipulation

- Adding or deleting clauses to the database during execution can be done by use of the following built-in predicates:
  - assert(C):
    Always succeeds and, as a side effect, causes a clause C to be added to the database.
  - retract(C):
    Deletes a clause that matches C from the database.
  - asserta(C):
    Adds C at the beginning of the database.
  - assertz(C):
    Adds C at the end of the database.

Repeat

- The built-in predicate `repeat` always succeeds, and each time it is reached by backtracking it generates a new execution branch.
- It behaves as if defined by:

  -
    ```
    repeat.
    repeat :- repeat.
    ```
- Example of use:

  -
    ```
    dosquares :-
        repeat,
        read(X),
        (X = stop, !;
         Y is X*X, write(Y), fail ).
    ```

bagof and setof

- Backtracking can generate all the objects that satisfy some goal. But when we generate a new solution, the previous one is lost. The built-in predicates `bagof`, `setof` and `findall` collect these solutions into a list.
- bagof(X,P,L):
  produces the list L of all the objects X such that a goal P is satisfied. This is useful only if X and P have some variables in common.
- setof(X,P,L):
  the same as bagof, but the elements in the list are ordered and without duplicates.

- Examples:

  -
    ```
    class( f, con).
    class( e, vow).
    class( d, con).
    class( c, con).
    class( b, con).
    class( a, vow).
    ```

  -
    ```
    bagof(Letter, class(Letter, con), List).
    > List = [f,d,c,b]
    ```

  -
    ```
    bagof(Letter, class(Letter,Class), List).
    > Class = con
      List = [f,d,c,b];

    > Class = vow
      List = [e,a]
    ```

-
  ```
  setof(Letter, class(Letter,con), List).
  > Letter = _0
    List = [b,c,d,f]

  setof((C1,Let), class(Let,C1), List).
  > C1 = _0
    Let = _1
    List = [(con,b),(con,c),(con,d),(con,f),
  (vow,a),(vow,e)]
  ```