Introduction to Prolog

References:

Reviews
- A Prolog program consists of predicate definitions.
- A predicate denotes a property or relationship between objects.
- Definitions consist of clauses.
- A clause has a head and a body (Rule) or just a head (Fact).
- A head consists of a predicate name and arguments.
- A clause body consists of a conjunction of terms.
- Terms can be constants, variables, or compound terms.
- We can set our program goals by typing a command that unifies with a clause head.
- A goal unifies with clause heads in order (top down).
- Unification leads to the instantiation of variables to values.
- If any variables in the initial goal become instantiated this is reported back to the user.

Tests
- When we ask Prolog a question we are asking for the interpreter to prove that the statement is true.
  ?- 5 < 7, integer(bob).
  yes = the statement can be proven.
  no = the proof failed because either
    - the statement does not hold, or
    - the program is broken.
  Error = there is a problem with the question or program.
  "nothing" = the program is in an infinite loop.
- We can ask about:
  - Properties of the database: mother(jane,alan).
  - Built-in properties of individual objects: integer(bob).
  - Relationships between objects:
    - Unification
    - Arithmetic relationships: <, >=, =, +, -

Arithmetic Operators
- Operators for arithmetic and value comparisons are built-in to Prolog
  = always accessible / don't need to be defined
- Comparisons: <, >, =<, =>, =:=, =\= (not equals)
  = Infix operators: go between two terms.
  =\= is used
    - 5 =\= 7. (infix)
    - =\=(5,7). (prefix)
    - all infix operators can also be prefixed
- Equality is different from unification
  = checks if two terms unify
  =\= compares the arithmetic value of two expressions
? - X = 5+4.  ? - X =\= 9.  yes  yes
- Mathematical precedence is preserved: /, *, before +, -
- Can make compound sums using round brackets
  - Impose new precedence
    |?- X = (5+4)*2.
    X = 18 ?
    yes
  - Inner-most brackets first
    |?- X is 5+4*2.
    X = 5+4 ?
    X = 9 ?
    yes

Arithmetic Operators (2)
- Arithmetic Operators: +, -, *, /
  = Infix operators but can also be used as prefix.
  - Need to use is to access result of the arithmetic expression otherwise it is treated as a term:
    |?- X = 5+4.
    X = 5+4 ?
    X = 9 ?
    yes
    (Can X unify with 5+4?)  (What is the result of 5+4?)
  - Mathematical precedence is preserved: /, *, before +, -
  - Can make compound sums using round brackets
    - Impose new precedence
      |?- X is (5+4)*2.
      X = 18 ?
      yes
    - Inner-most brackets first
      |?- X = 5+4*2.
      X = 9 ?
      yes

Unify with occurs check
- |- X=X Y.
  Y = X
  yes
- |- X = f(Y,Y).
  X = f(Y,Y)
  yes
- |- X = f(X,Y).
  cannot display cyclic term for X
  |?- unify(X,f(X,Y)).
  uncaught exception:
  error(existence_error(procedure,unify/2),top_level/0)
- |- unify_with_occurs_check(X, f(X,Y)).
  no
Tests within clauses

- These operators can be used within the body of a clause:
  - To manipulate values:
    ```prolog
    sum(X,Y,Sum):-
    Sum is X+Y.
    ```
  - To distinguish between clauses of a predicate definition:
    ```prolog
    bigger(X,Y,Z):-
    X < Y, write('The bigger number is '), write(Z).
    bigger(X,Y,Z):-
    X > Y, write('The bigger number is '), write(Z).
    bigger(X,Y,Z):-
    X =:= Y, write('Numbers are the same').
    ```

Backtracking

|?- bigger(5,4).
|-----
|5 < 4, fails
|write('The bigger number is '), write(5).

Reaches full-stop
The bigger number is 5
yes
|?-
Backtracking

?- bigger(5,5).
\rightarrow If our query only matches the final clause.
\rightarrow If bigger(N,M):
\rightarrow \begin{cases} \text{N < M, write('The bigger number is '), write(M).} \\
\text{N > M, write('The bigger number is '), write(N).} 
\end{cases}
\rightarrow Satisfies the same conditions.
\rightarrow write('Numbers are the same')).

Clues should be ordered according to specificity.
Most specific at top. Universally applicable at bottom.

Numbers are the same.

Reporting Answers

?- bigger(5,4).
\rightarrow Question is asked.
The bigger number is 5.
\rightarrow Answer is written to terminal.
\rightarrow Succeeds but answer is lost.

- This is fine for checking what the code is doing but not for using the proof.

?- bigger(6,4), bigger(Answer,5).
\rightarrow Instantiation error!

- To report back answers we need to:
  - put an uninstantiated variable in the query,
  - instantiate the answer to that variable when the query succeeds,
  - pass the variable all the way back to the query.

Passing Back Answers

- To report back answers we need to:
  1. put an uninstantiated variable in the query,

?- max(6,4,Answer), max(Answer,5,New_answer).

2. instantiate the answer to that variable when the query succeeds,

3. pass the variable all the way back to the query.

Head Unification

- To report back answers we need to:
  1. put an uninstantiated variable in the query,

?- max(6,4,Answer), max(Answer,5,New_answer).

Or, do steps 2 and 3 in one step by naming the variable in the head of the clause the same as the correct answer.
\rightarrow head unification

Satisfying Subgoals

- Most rules contain calls to other predicates in their body. These are known as Subgoals.
- These subgoals can match:
  - facts,
  - other rules, or
  - the same rule = a recursive call

1) drinks(alan,beer).
2) likes(alan,coffee).
3) likes(heather,coffee).
4) likes(Person,Drink):-
\rightarrow \text{Drinks(Person,Drink).} \rightarrow a different subgoal
5) likes(Person,Drinks):
\rightarrow \text{likes(Person,Drinks), recursive subgoals}
\rightarrow \text{likes(Somebody,Drink), Likely loop forever}

Central Ideas of Prolog

- SUCCESS/FAILURE
  - any computation can "succeed" or "fail", and this is used as a test mechanism.

- UNIFICATION (2-WAY MATCHING)
  - any two data items can be compared for similarity, and values can be bound to variables in order to allow a match to succeed.

- SEARCHING
  - the whole activity of the Prolog system is to search through various options to find a combination that succeeds.
    - Main search tools are backtracking and recursion.

- BACKTRACKING
  - when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.
Why Use Recursion?

- It allows us to define very clear and elegant code.
  - Why repeat code when you can reuse existing code.
- Relationships may be recursive
  - e.g. "X is my ancestor if X is my parent’s ancestor."
- Data is represented recursively and best processed iteratively.
  - Ordered data: each element requires the same processing
- Allows Prolog to perform complex search of a problem space without any dedicated algorithms.

Prolog Data Objects (Terms)

- Simple objects
  - Constants
  - Integers
  - Atoms
  - Symbols
  - Strings
  - Signs

- Structured objects
  - Variables
  - Structures
  - Lists

- Example:
  - date(10,04)
  - person(bob,48)
  - \[a,b\]
  - \[a\]

Structures

- To create a single data element from a collection of related terms we use a structure.
- A structure is constructed from a function name (functor) and one or more components.
- The components can be of any type: atoms, integers, variables, or structures.
- As functors are treated as data objects just like constants they can be unified with variables

```
?- X = date(03,31,05).
X = date(03,31,05)
```

Lists

- A collection of ordered data.
- Has zero or more elements enclosed by square brackets ([ ]) and separated by commas (,).
- Like any object, a list can be unified with a variable

```
?- X = [any, list, of elements].
X = [any, list, of elements]?
```

List Unification

- Two lists unify if they are the same length and all their elements unify.
- The bar notation turns everything after the Head into a list and unifies it with Tail.

```
|?-(a,b,c,d)=\([a,b,c,d]\). |
|?-(\([a,X]\),\([Y+b]\))=\([a+c],\([d+b]\)).
A = a
B = b
C = c
D = d
yes
```

Definition of a List

- Lists are recursively defined structures.
- An empty list, [], is a list.
- A structure of the form \([X, \ldots]\) is a list if \(X\) is a term and \([\ldots]\) is a list, possibly empty.
- This recursiveness is made explicit by the bar notation
  - \([\text{Head}|\text{Tail}]\).
- **Head** must unify with a single term.
- **Tail** unifies with a list of any length, including an empty list, [].
  - the bar notation turns everything after the Head into a list and unifies it with Tail.

```
|?-(\([X,a]\)=\([b,Y]\). |
|?-(\([a]\),\([b,c]\),\([\ldots]\)=\([a]\),\([b,c]\),\([\ldots]\).
X = [a]
Y = []
yes
```
Head and Tail

\[ [a,b,c,d] = \{ \text{Head} | \text{Tail} \} \]

\[ X = a, \]
\[ Y = b, \]
\[ Z = \{ c,d \} \]

\[ [a] = \{ \text{Head} | \text{Tail} \} \]
\[ W = a, \]
\[ X = \{ \} \]
\[ Y = c, \]
\[ Z = \{ \} \]

\[ [\{ \} ] = \{ \text{Head} | \text{Tail} \} \]
\[ \text{List} = \{ a,b,c \} \]

Identifying a list

- We said that lists are recursively defined structures:
  
  “An empty list, [], is a list.
  A structure of the form [X, ...] is a list if X is a term and [...] is a list, possibly empty.”

- This can be tested using the Head and Tail notation, [H|T], in a recursive rule.

  \[ \text{is}_a\_\text{list}([[]]). \]
  \[ \text{is}_a\_\text{list}(\text{List}). \]

Head is replaced with an underscore as we don’t care its name.

Focussed Recursion

- To ensure that the predicate terminates, the recursive case must move the problem closer to a solution.
  - If it doesn’t it will loop infinitely.
  - With list processing this means stripping away the Head of a list and recursing on the Tail.

  \[ \text{is}_a\_\text{list}(\{\}). \]
  \[ \text{is}_a\_\text{list}(\text{List}). \]

- The same focussing has to occur when recursing to find a property or fact.

  \[ \text{is}_\text{older}(\text{Ancestor}, \text{Person}). \]
  \[ \text{is}_\text{older}(\text{Someone}, \text{Person}). \]
  \[ \text{is}_\text{older}(\text{Ancestor}, \text{Someone}). \]

Focussed Recursion (2)

Given this program:

\[ \text{parent}(\text{tom}, \text{jim}). \]
\[ \text{parent}(\text{mary}, \text{tom}). \]
\[ \text{is}_\text{older}(\text{Old}, \text{Young}). \]

- A query looking for all solutions will loop.

\[ ?-\text{is}_\text{older}(\text{K}, \text{Y}). \]
\[ X = \text{tom}, \]
\[ Y = \text{jim} ; \]
\[ X = \text{mary}, \]
\[ Y = \text{tom} ; \]
\[ X = \text{mary}, \]
\[ Y = \text{jim} ; \]

It loops because the recursive clause does not focus the search it just splits it. If the recursive is_older doesn’t find a parent it just keeps recursing on itself.

Focussed Recursion (3)

The correct program:

\[ \text{parent}(\text{tom}, \text{jim}). \]
\[ \text{parent}(\text{mary}, \text{tom}). \]
\[ \text{is}_\text{older}(\text{Old}, \text{Young}). \]

- Can generate all valid matches without looping.

\[ ?-\text{is}_\text{older}(\text{K}, \text{Y}). \]
\[ X = \text{tom}, \]
\[ Y = \text{jim} ; \]
\[ \text{is}_\text{older}(\text{Old}, \text{Young}). \]

To make the problem space smaller we need to check that Young has a parent before recursion. This way we are not looking for something that isn’t there.
Quick Aside: Tracing Prolog

- To make Prolog show you its execution of a goal type `trace`, at the command line.
  - Prolog will show you:
    - which goal is called with which arguments,
    - whether the goal succeeds (Exit),
    - has to be Redone, or Fails.
  - The tracer also indicates the level in the search tree from which a goal is being called.
    - The number next to the goal indicates the level in the tree (top level being 0).
    - The leftmost number is the number assigned to the goal (every new goal is given a new number).
- To turn off the tracer type `notrace`.

Tracing Member/2

```
?- trace.
?- member(ringo,[john,paul,ringo,george]).
1 1 Call: member(ringo,[john,paul,ringo,george]) ?
2 2 Call: member(ringo,[paul,ringo,george]) ?
3 3 Call: member(ringo,[ringo,george]) ?
3 3 Exit: member(ringo,[ringo,george]) ?  
2 2 Exit: member(ringo,[paul,ringo,george]) ?
1 1 Exit: member(ringo,[john,paul,ringo,george]) ?  yes
```

```
?- member(stuart,[john,paul,ringo,george]).
1 1 Call: member(ringo,[john,paul,ringo,george]) ?
2 2 Call: member(ringo,[paul,ringo,george]) ?
3 3 Call: member(ringo,[ringo,george]) ?
4 4 Call: member(stuart,[george]) ?
5 5 Call: member(stuart,["]) ?  \[\] does not match \[H|T\]
5 5 Fail: member(stuart,["]) ?
4 4 Fail: member(stuart,[george]) ?
3 3 Fail: member(ringo,[ringo,george]) ?
2 2 Fail: member(ringo,[paul,ringo,george]) ?
1 1 Fail: member(ringo,[john,paul,ringo,george]) ?  no
```

Prolog’s Persistence

- When a sub-goal fails, Prolog will backtrack to the most recent successful goal and try to find another match.
- Once there are no more matches for this sub-goal it will backtrack again; retrying every sub-goal before failing the parent goal.
- A call can match any clause head.
- A redo ignores old matches.

```
a:- b, c, d, e, f, g, h, i, j.
```

```
a:- b, c, d, e, f, g, h, i, j.
```

Failing the parent goal

```
a:- b, c, d, e, f, g, h, i, j.
a:- k.
a:- m.
```

- The cut succeeds when it is called and commits the system to all choices made between the time the parent goal was invoked and the cut.
- This includes committing to the clause containing the cut.
- The goal can only succeed if this clause succeeds.
- When an attempt is made to backtrack through the cut
  - the clause is immediately failed, and
  - no alternative clauses are tried.

Mutually Exclusive Clauses

- We should only use a cut if the clauses are mutually exclusive (if one succeeds the others won’t).
- If the clauses are mutually exclusive then we don’t want Prolog to try the other clauses when the first fails = redundant processing.
- By including a cut in the body of a clause we are committing to that clause.
  - Placing a cut at the start of the body commits to the clause as soon as head unification succeeds.
  - Placing a cut somewhere within the body (even at the end) states that we cannot commit to the clause until certain sub-goals have been satisfied.

```
a(1,X):- k, b(X), c(X).
a(_,X):- b(X), c(X), !.
```

Cut !

- If we want to restrict backtracking we can control which sub-goals can be redone using the cut `!`.
- We use it as a goal within the body of clause.
- It succeeds when called, but fails the parent goal (the goal that matched the head of the clause containing the cut) when an attempt is made to redo it on backtracking.
- It commits to the choices made so far in the predicate.
  - unlimited backtracking can occur before and after the cut but no backtracking can go through it.

```
a:- b, c, d, e, f, g, h, i, j.
a:- k.
a:- m.
```

This clause and these choices committed to
Mutually Exclusive Clauses (2)

\[ f(X,0) \text{:- } X < 3. \]
\[ f(X,1) \text{:- } 3 < X, X < 6. \]
\[ f(X,2) \text{:- } 6 < X. \]

\[ ?- \text{trace, } f(2,N). \]
\[ 1 \text{ Call: } f(2,0) ? \]
\[ 2 \text{ Call: } 2 < 3 ? \]
\[ 2 \text{ Exit: } 2 < 3 ? \]
\[ 1 \text{ Exit: } f(2,0) ? \]
\[ N = 0 ? ; \]
\[ 1 \text{ Redo: } f(2,0) ? \]

Green Cuts

\[ f(X,0) \text{:- } X < 3, !. \]
\[ f(X,1) \text{:- } 3 <= X, X < 6, !. \]
\[ f(X,2) \text{:- } 6 <= X. \]

\[ ?- \text{trace}, f(2,N). \]
\[ 1 \text{ Call: } f(2,0) ? \]
\[ 2 \text{ Call: } 2 < 3 ? \]
\[ 2 \text{ Exit: } 2 < 3 ? \]
\[ 1 \text{ Exit: } f(2,0) ? \]
\[ N = 0 ? ; \]

If you reach this point don't bother trying any other clause.

- Notice that the answer is still the same, with or without the cut.
  - This is because the cut does not alter the logical behaviour of the program.
  - It only alters the procedural behaviour: specifying which goals get checked when.

- This is called a green cut. It is the correct usage of a cut.

- Be careful to ensure that your clauses are actually mutually exclusive when using green cuts!

Red Cuts

\[ f(X,0) \text{:- } X < 3. \]
\[ f(X,1) \text{:- } X < 6. \]
\[ f(X,2). \]

\[ ?- \text{f(7,N).} \]
\[ 1 \text{ Call: } f(7,0) ? \]
\[ 2 \text{ Call: } 7 < 3 ? \]
\[ 2 \text{ Fail: } 7 < 3 ? \]
\[ 3 \text{ Call: } 7 < 6 ? \]
\[ 3 \text{ Fail: } 7 < 6 ? \]
\[ 1 \text{ Exit: } f(7,2) ? \]
\[ N = 2 ? \]
\[ yes \]

- Red cuts change the logical behaviour of a predicate.
- TRY NOT TO USE RED CUTS!
- Red cuts make your code hard to read and are dependent on the specific ordering of clauses (which may change once you start writing to the database).
- If you want to improve the efficiency of a program use green cuts to control backtracking.

Using the cut

- Red cuts change the logical behaviour of a predicate.
- TRY NOT TO USE RED CUTS!

A larger example.

We'll define several versions of the disjoint partial map split.

\[ \text{split}([\text{integers}], \text{non-negatives}, \text{negatives}). \]

1. A version not using cut. Good code (each can be read on its own as a fact about the program). Not efficient because choice points are retained. The first solution is desired, but we must ignore backtracked solutions.

\[ \text{split}([\text{integers}], [\text{non-negatives}, \text{negatives}]). \]
\[ \text{split}([\text{integers}], [\text{non-negatives}, \text{negatives}]). \]
\[ \text{split}([\text{integers}, \text{non-negatives}, \text{negatives}]). \]

- Do not use cuts in place of tests.
To ensure a logic friendly cut either:

\[ p(X) :\text{- test}(X), !, \text{call}(X). \]
\[ p(X) :\text{- test}(X), !, \text{call}(X). \]
\[ p(X) :\text{- test}(X), !, \text{call}(X). \]
\[ p(X) :\text{- test}(X), !, \text{call}(X). \]

- The mutually exclusive tests are in the head of the clause.

Redundant? Yes

- Because the clauses are mutually exclusive and ordered we know that once the clause above fails certain conditions must hold.
- We might want to make our code more efficient by removing superfluous tests.
A larger example.

2. A version using cut. Most efficient, but not the best “defensively written” code. The third clause does not stand on its own as a fact about the problem. As in normal programming languages, it needs to be read in context. This style is often seen in practice, but is deprecated.

```
split([], [], []).  
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).  
split([H|T], R, [H|Z]) :- split(T, R, Z).
```

Minor modifications (adding more clauses) may have unintended effects. Backtracked solutions invalid.

A larger example.

3. A version using cut which is also “safe”. The only inefficiency is that the goal H < 0 will be executed unnecessarily whenever H < 0.

```
split([], [], []).  
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).  
split([H|T], R, [H|Z]) :- H < 0, split(T, R, Z).
```

Recommended for practical use. Hidden problem: the third clause does not capture the idea that H < 0 is a committal. Here committal is the default because H < 0 is in the last clause. Some new compilers detect that H < 0 is redundant.

Negation and the Cut

- The Cut ‘!’
  - A device for controlling the search
  - Used to increase efficiency
  - BUT can alter semantics of a program — change its solution set.
- Negation as Failure
  - Negation succeeds if search fails.
  - Not Constructive - Unification does not produce any bindings.
  - Consistent Interpretation depends on Closed World Assumption

Negation as Failure

Using cut together with the built-in predicate fail, we may define a kind of negation. Examples:

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

A utility predicate meaning something like "not equals":

```
different(X, X) :- 1, fail.  
different(_, _).
```

We can use the same idea of “cut fail” to define the predicate not, which takes a term as an argument, not will “call” the term, that is evaluate it as though it is a goal:

```
not(G) :- call(G), !, fail.  
not(_).
```

call is a built-in predicate.
Negation as Failure

Most Prolog systems have a built-in predicate like not. GProlog calls it \+. Remember, not does not correspond to logical negation, because it is based on the success/failure of goals. It can, however, be useful:

\[ \text{likes(mary, X)} :- \neg \neg (\text{reptile}(X)). \]
\[ \text{different(X, Y)} :- \neg (X = Y). \]

\[ \neg \neg \text{married}(X), \text{student}(X). \]
\[ \text{student}(\text{bill}). \]
\[ \text{student}(\text{joe}). \]
\[ \text{married}(\text{joe}). \]

?- single_student(\text{bill}).
\text{yes.}

?- single_student(\text{joe}).
\text{no.}

?- single_student(\text{bill}).
\text{yes.}

?- single_student(\text{joe}).
\text{no.}

?- single_student(\text{bill}).
\text{no.}

?- single_student(\text{joe}).
\text{no.}

Negation as Failure 2nd Try

\[ \text{single_student}(X) :- \]
\[ \text{student}(X), \neg \neg \text{married}(X). \]
\[ \text{student}(\text{bill}). \]
\[ \text{student}(\text{joe}). \]
\[ \text{married}(\text{joe}). \]

?- single_student(\text{bill}).
\text{yes.}

?- single_student(\text{joe}).
\text{no.}

?- single_student(\text{bill}).
\text{yes.}

?- single_student(\text{joe}).
\text{no.}

?- single_student(X)
\text{no.}

?- single_student(X)
\text{no.}

?- single_student(X)
\text{X=bill.}

Negation-by-failure can be non-logical

Some disturbing behavior even more subtle than the innocuous problem, and can lead to some extremely obscure programming errors. Here is a restaurant database:

\[ \text{good_standard}(\text{goedels}). \]
\[ \text{good_standard}(\text{hilberts}). \]
\[ \text{expensive}(\text{goedels}). \]
\[ \text{reasonable}(R) :- \neg \neg (\text{expensive}(R)). \]

Consider the following dialogue:

?- \text{good_standard}(X), \text{reasonable}(X).
\text{X=hilberts}

But if we ask the logically equivalent question:

?- \text{reasonable}(X), \text{good_standard}(X).
\text{no.}

X = \text{hilberts}
Question
Why do we get different answers for what seem to be logically equivalent queries?

The difference between the questions is as follows.
In the first question, the variable X is always instantiated when reasonable(X) is executed.
In the second question, X is not instantiated when reasonable(X) is executed.
The semantics of reasonable(X) differ depending on whether its argument is instantiated.

Not a Good Idea!
It is bad practice to write programs that destroy the correspondence between the logical and procedural meaning of a program without any good reason for doing so.
Negation-by-failure does not correspond to logical negation, and so requires special care.

How to fix it?
One way is to specify that negation is undefined whenever an attempt is made to negate a non-ground formula.
A formula is ‘ground’ if it has no unbound variables.

Some Prolog systems issue a run-time exception if you try to negate a non-ground goal.

Clauses and Databases
In a relational database, relations are regarded as tables, in which each element of an n-ary relation is stored as a row of the table having n columns.

supplier(jones, chair, red, 10)
supplier(smith, desk, black, 50)

Using clauses, a table can be represented by a set of unit clauses. An n-ary relation is named by an n-ary predicate symbol.
supplier(jones, chair, red, 10).
supplier(smith, desk, black, 50).

Clauses and Databases
Advantages of using clauses:
1. Rules as well as facts can coexist in the description of a relation.
2. Recursive definitions are allowed.
3. Multiple answers to the same query are allowed.
4. There is no role distinction between input and output.
5. Inference takes place automatically.

Negation and Representation
Like databases, clauses cannot represent negative information. Only true instances are represented.

The battle of Waterloo occurred in 1815.

How can we show that the battle of Waterloo did not take place in 1923? The database cannot tell us when something is not the case, unless we do one of the following:
1. ‘Complete’ the database by adding clauses to specify the battle didn’t occur in 1814, 1813, 1812, ..., 1816, 1817, 1818, ...
2. Add another clause saying the battle did not take place in another year (the battle occurred in and only in 1815).
3. Make the ‘closed world assumption’, implemented by ‘negation by failure’.
Closed World Assumption

- Assumption that the world is defined in its entirety
- The representation is “complete”/“closed”
- No true statement is missing from the representation
- In practice, assumed for conventional databases
  - “Sorry, Sir, you must NOT exist, since your social security number is NOT IN our database, bye, bye.”
- From a logic program, P, allows us to conclude
  - the negation of A, IF A is NOT IN the meaning of P

Negation as Failure & the CWA

single_student(X) :-
  student(X),
  (\+ married(X)).
student(bill).
student(joe).
married(joe).
student(jim).
?- single_student(bill).
  yes.
?- single_student(joe).
  no.
?- single_student(jim).
  yes.

But Jim IS married.
Maybe I should read up on the CWA.

The Cut (!)

- The one and only ‘!’
  - There are GOOD, BAD and Ugly ones (usages).
  - GREEN and RED ones (usages).
- Goals before a cut produce first set and only the first set of bindings for named variables
  - Commits a choice
  - No alternative matches considered upon backtracking.
- Green Cuts
  - Exclude clauses (solution attempts), but NOT solutions.
  - Removal of Cut does NOT change the meaning of the program. The cut’s positioning just effects efficiency.
- Red Cuts
  - Alter the actual meaning of the program.

A Good Red Cut

if_then_else(If,Then,Else) :-
  If, !, Then.
if_then_else(If,Then,Else) :-
  Else.
?- if_then_else(true, write(equal), write(not_equal))
equal
  yes.
?- if_then_else(false, write(equal), write(not_equal))
not_equal
  yes.

If we take out the cut we change the meaning -- so the cut is RED.
But it is used to produce the meaning we want -- so the cut is GOOD.

A Bad Red cut

- \( \text{min}(N1,N2,N1) \) \( \cdot \) \( N1 < N2 \).
- \( \text{min}(N2,N2) \).

A BAD Red Cut

The cut is used to implement the default case -- Yike!

R1. pension(X,disabled) :- disabled(X),!.
R2. pension(X,senior) :- over65(X), paid_up(X),!.
R3. pension(X,supplemental) :- over65(X),!.
R4. pension(X,nothing). %"The Default" If everything else fails.
F1. disabled(joe).
F2. over65(joe).
F3. paid_up(joe).
Q1. ?- pension(joe, nothing). \rightarrow yes.
OOPS! "I'm sorry Mr. Joe...yes Mr. Joe you are entitled, it was a small computer error, really. Mr. Joe, computers DO make mistakes...I'm sorry what was that about intended meaning?"
Q2. ?- pension(joe, P). \rightarrow P = disabled
Does he get more than one pension payment?
Q3. ?- pension(X, senior) \rightarrow X = joe
What happened to Lou's pension? Isn't he a senior?
Joe's Revenge

R1. pension(X, disabled_pension) :- disabled(X).
R2. pension(X, senior_pension) :- over65(X), paid_up(X).
R3. pension(X, supplemental_pension) :- over65(X).
R4. entitled(X, Pension) :- pension(X, Pension).
R5. entitled(X, nothing) :- ! (pension(X, Pension)).

F1. disabled(joe).
F2. over65(joe).
F3. paid_up(joe).

Q1. ?- entitled(joe, nothing) -> no.
Q2. ?- entitled(joe, P) :- 1. P = disabled, 2. P = senior, 3. P = supplemental
Q3. ?- entitled(X, senior_pension) :- 1. X = joe, 2. X = lou.
Q4. ?- entitled(X, disabled_pension) :- 1. X = joe.

Summary

• Controlling backtracking: the cut !
  – Efficiency: avoids needless REDO-ing which cannot succeed.
  – Simpler programs: conditions for choosing clauses can be simpler.
  – Robust predicates: definitions behave properly when forced to REDO.

• Green cut = cut doesn't change the predicate logic = good
• Red cut = without the cut the logic is different = bad
• Cut – fail: when it is easier to prove something is false than true.