Utrecht University

Introduction to Prolog 4

Expert Systems

Alvaro H. C. Correia

13th of June 2019

Utrecht University

Recap

Facts, Rules and Queries

Prolog Statements

Facts state what is always true.

```
man(socrates).
```

Rules state what is true if some conditions hold.

```
mortal(X) :- man(X).
```

Queries are how we interact with the program.

They allows us to check what logically follows from our facts and rules.

```
?- mortal(socrates).
true.
```

Rules

Conjunction

```
head :- goal_1, goal_2, ..., goal_n.
```

Disjunction - Procedure

```
head :- goal_1.
head :- goal_2.
...
head :- goal_n.
```

Facts, Rules and Queries

Matching

Two terms match if

- They are identical;
- Their variables can be instantiated so as to make them identical.

Search

- Prolog tries to match each goal in a query **in order**.
- If one of the goals fails, it backtracks to the **last** choicepoint.
- A choicepoint is a previous goal where more than one variable instantiation produce a match.
- The search process can be seen as a **tree**.

Recursion

There is no explicit loop in Prolog.

No while, until or for.

Repetition by recursion.

- Call the same predicate as condition.
- Stop when a special or base case is reached.

Base and special cases usually come first!

Procedural vs. **Declarative** meanings

Manipulating lists

Going through a list

1. Define our base case: the empty list.
2. Get the value of the head.
3. Call the same predicate recursively on the tail.

```
printList([]).
printList([H|T]):-
    writeln(H),
    printList(T).
```

Manipulating lists

Returning a list

1. The base case compares two empty lists.
2. Match the values of both heads.
3. Call the same predicate recursively on the tails.

```
returnList([], []).
returnList([H|T], [H2|T2]):-
    doSomething(H, H2)
    returnList(T, T2).
```

Manipulating lists

Returning a list

1. The base case compares two empty lists.
2. Match the values of both heads.
3. Call the same predicate recursively on the tails.

For instance, we could copy a list.

```
returnList([], []).
returnList([H|T], [H2|T2]):-
    H = H2,
    returnList(T, T2).
```

```
?-returnList([1,2,3], L).
L= [1,2,3].
```

Manipulating lists

Built-in Predicates

member/2

```
?- member(X, [1,2,3]).
X = 1;
```

append/3

```
?- append([1], [2,3], L).
L = [1,2,3].
```

length/2

```
?- length([1,2,3], L).
L = 3.
```

sort/2

```
?- sort([2, 1, 3], L).
L = [1,2,3].
```

Negation \+

Negation as failure

In Prolog, negation is defined via its failure to provide a proof.

If it does not follow from the database, it is false!

Closed-world assumption.

\+ operator

Make the success of a goal dependent on the failure of a **subgoal**.

We cannot use negation in the head of a rule!

Cut!

!

Cut is a special operator represented by the exclamation sign.

- Always succeeds.
- Discards all previous choicepoints.

  No alternative solutions for already instantiated variables will be considered.
- Reduces computational costs.
- Avoids undesirable solutions.

We use cut to prune the search tree, **changing the procedural meaning**.

Green Cut	Same declarative meaning.
Red Cut	Different declarative meaning.

Dynamic Predicates

Updating the database

We can update clauses dynamically during the execution of the program

```
?- dynamic(foo/1). % declare foo as a dynamic predicate.

?- assertz(foo(100)). % add to the end of the database.

?- asserta(foo(1)). % add to the beginning of the database.

?- listing(foo). % show all clauses of predicate foo.
    :- dynamic foo/1.
    foo(1).
    foo(100).

?- retractall(foo(_)). % remove all clauses of predicate foo.

?- listing(foo).
    :- dynamic foo/1.
```

Control Predicates

Control Predicates

Control predicates coordinate how the program runs

They define the procedural and declarative meanings of the program.

fail / **false**	Always fails.
true	Always succeeds.
,	Conjunction. Logical and.
;	Disjunction. Logical or.
!	Cut. Prunes the search tree.
\+	Negation (as failure).
->	if -> then
***->**	soft cut (rarely used)
repeat	repeats goals until succeeding.

-> Operator

If -> Then ; Else

X -> Y.

If X can be satisfied, attempt Y.

X -> Y ; Z.

If X can be satisfied, attempt Y, otherwise attempt Z.

The -> operator commits to the choices made on its left-hand side.

- We only consider the first solution for X.
- Backtracking can generate other solutions for Y and Z, but not for X.
- Avoid! We can always rewrite the code in simpler clauses without ->.

-> Operator - Examples

Calculating the factorial function

Consider the factorial example from the second lecture.

```
fac(0, 1) :- !. % base case
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

-> Operator - Examples

Calculating the factorial function

Notice also that we added a cut to the base case. Why is that a good idea?

```
fac(0, 1) :- !. % base case
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

```
?- fac(5, F)
F=120.
?- fac(6, F)
F=720.
?- fac(5, F)
F=5040.
```

-> Operator - Examples

Calculating the factorial function

We can rewrite the same predicate using the -> operator.

```
fac(N, F) :-
    N > 0,
    (N=0 -> F=1
        ;
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx).
```

```
?- fac(5, F)
F=120.
?- fac(6, F)
F=720.
?- fac(5, F)
F=5040.
```

-> Operator - Examples

Calculating the factorial function

The if-then-else case replaces the base the case.

```
fac(N, F) :-
    N > 0,
    (N=0 -> F=1
        ;
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx).
```

```
fac(0, 1) :- !.
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

Give preference to writing short clauses, as in the original example.

They are clearer and closer to the declarative meaning of the program.

repeat

repeat/0

`repeat` always succeeds and provides an infinite number of choicepoints.

Whenever the program fails, we eventually return to `repeat`.

repeat

repeat/0

Because of the infinite choicepoints, we always return to repeat after failing.

```
persistent :-
    repeat,
    writeln("Should I stop?"),
    read(S),
    S == yes,
    writeln("Goodbye!").
```

```
?- persistent.
Should I stop?
?- no.
Should I stop?
?- no.
Should I stop?
?- yes.
Goodbye!
true.
```

repeat

repeat/0

`repeat` is actually implemented in two simple clauses.

```
repeat.
repeat :- repeat.
```

It is possible to create a custom repeat predicate.

Not a good coding practice but a good exercise.

```
customRepeat.
customRepeat :-
    writeln("repeating"),
    customRepeat.
```

Input and Output

Streams

In Prolog the interface with the user is managed via streams.

Input Stream

Where Prolog reads queries and programs (clauses).

The default input stream is the terminal screen.

Output Stream

Where Prolog writes the solution to queries.

The default output stream is also the terminal screen.

Writing to the Output Stream

Built-in predicates

Useful for debugging or requesting information from the user.

write/1, **writeln/1** and **print/1**

```
?- write("How can I help you?").
How can I help you?
?- print("Thanks").
"Thanks"
```

format/2

```
?- format('~w~46t~w~46t~w~72|~n', [1, 2, 3]).
1.......................2.......................3
?- X = 0, format('~w~46t~w~46t~w~72|~n', [X, 2, 3]).
0.......................2.......................3
```

Changing the Output Stream - Writing to a file

Built-in predicates tell/1 and told/0.

1. `tell/1` specifies the output stream.
   - `tell(myfile)` sets the output to a .txt named ”myfile”.
     - If there is no file named ”myfile”, a new file is created.
     - If there is a file named ”myfile”, it is overwritten.
   - `tell(user)` sets the output to the terminal screen (default).
2. Anything written will be redirected to the new output stream.
3. Once we finished writing we close the file with `told`.

```
?- tell(out).
?- write(”Now we are writing to a file.”)
?- told.
```

Changing the Output Stream - Writing to a file

Built-in predicates append/1 and told/0.

1. `append/1` specifies the output stream.
   - `append(myfile)` sets the output to a .txt named ”myfile”.
     - If there is no file named ”myfile”, a new file is created.
     - If there is a file named ”myfile”, we **append** to it.
   - `append(user)` sets the output to the terminal screen (default).
2. Anything written will be **appended** to the new output stream.
3. Once we finished writing we close the file with `told`.

```
?- append(out).
?- write(”We had forgotten to add something.”)
?- told.
```

Reading from the Input Stream

Getting input from the user

read(-Term)

- Wait for an input from the user.
- Bind Term to the term he/she has typed.
- Term is read as a Prolog statement.

```
copy :- read(Term),
    writeln(Term).
```

```
?- copy.
|: Variable.
_293
true.
```

Reading from the Input Stream

Getting input from the user

read_line_to_string(user_input, S)

- Wait for an input from the user.
- Bind S to the string he/she has typed.
- user_input is a built-in alias for user input stream.

```
copy :-
    read_line_to_string(user_input, S),
    writeln(S).
```

```
?- copy.
|: Variable.
Variable
true.
```

Reading from the Input Stream

Getting input from the user

read_line_to_codes(user_input, C).

- Wait for an input from the user.
- Bind C to the list of ASCII code characters he/she has typed.
- user_input is a built-in alias for user input stream.

```
copy :-
    read_line_to_codes(user_input, C),
    writeln(C).
```

```
?- copy.
|: Hoi.
C = [72, 111, 105].
true.
```

Changing the Input Stream - Reading from a file

Built-in predicates see/1 and seen/0

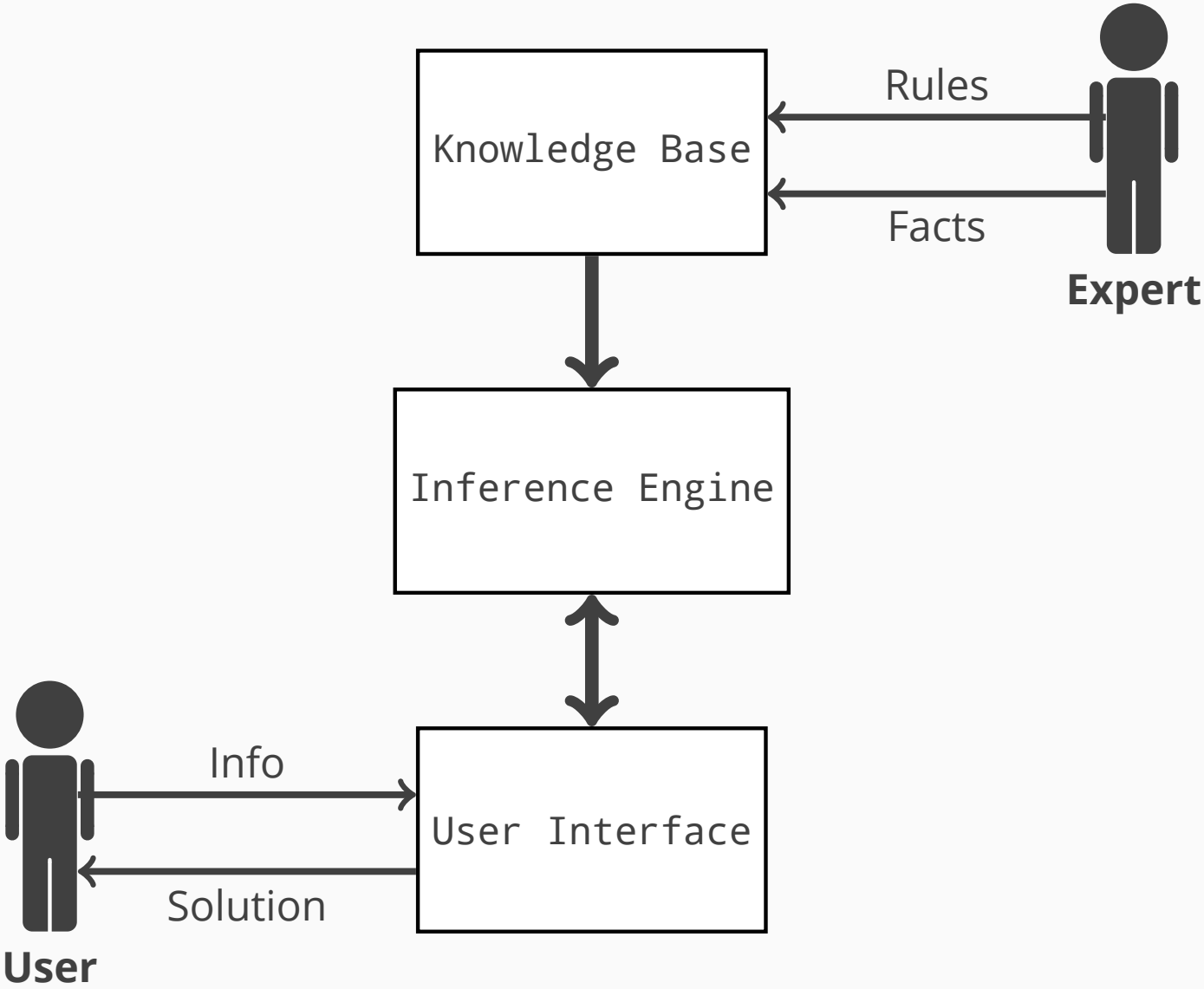
1. `see/1` specifies the input stream.
   - `see(myfile)` sets the input stream to a file named ”myfile”.
     - If there is no file named ”myfile”, Prolog throws an error.
   - `see(user)` sets the input stream to the terminal screen (default).
2. Each time we call `read/1`, Prolog will read until it finds a dot.
3. Once we have finished reading we close the file with `seen`.

```
% input file
1. 2.
```

```
?- see(input).
?- read(X), read(Y), Z is X + Y.
?- X = 1, Y = 2, Z = 3.
```

Expert Systems

Expert System



User Interface

User Interaction

We interact with the user in three main ways

1. Request input from the user
    1.1 Pose a question
    1.2 List options
2. Read and process input from the user
    2.1 Run inference
    2.2 Request further information
3. Return result from inference

Requesting User Input

We can define a menu predicate

1. Print out the options.

```
menu :-
    repeat,
    writeln("Menu"),
    writeln(”1. Option 1”),
    writeln(”2. Option 2”),
    writeln(”0. Exit”),
    read_line_to_string(user_input, Choice),
    process_choice(Choice),
    Choice == "0".
:- menu. % call menu as soon as file is loaded
```

Requesting User Input

We can define a menu predicate

2 Wait for the user input.

```
menu :-
    repeat,
    writeln("Menu"),
    writeln("1. Option 1"),
    writeln("2. Option 2"),
    writeln("0. Exit"),
    read_line_to_string(user_input, Choice),
    process_choice(Choice),
    Choice == "0".
:- menu. % call menu as soon as file is loaded
```

Requesting User Input

We can define a menu predicate

3 Process the user choice using another procedure.

```
menu :-
    repeat,
    writeln("Menu"),
    writeln("1. Option 1"),
    writeln("2. Option 2"),
    writeln("0. Exit"),
    read_line_to_string(user_input, Choice),
    process_choice(Choice),
    Choice == "0".
:- menu. % call menu as soon as file is loaded
```

Requesting User Input

We can define a menu predicate

4 If the choice is ”0”, the program fails and repeats.
Otherwise, it succeeds and terminates.

```
menu :-
    repeat,
    writeln("Menu"),
    writeln(”1. Option 1”),
    writeln(”2. Option 2”),
    writeln(”0. Exit”),
    read_line_to_string(user_input, Choice),
    process_choice(Choice),
    Choice == "0".
:- menu. % call menu as soon as file is loaded
```

Requesting User Input

Built-in predicate menu/3

`menu(+Title, +Options, -Choice)`

- Options is a list of terms `Choice:TextField`, where `Choice` is an atom.
- Cleans terminal screen and ignores invalid options.

```
:- repeat,
    menu('Menu', [
            1 : "Option 1",
            2 : "Option 2",
            0 : "Exit"
        ], Choice).
    process_choice(Choice),
    Choice == 0. % Choice is an atom not a string.
```

Process User Options

We create another predicate to process each option

`process_choice(+Choice)`

In the exit case, we simply succeed to exit the `repeat` loop.

```
process_user_choice("0"). % user choice is 0: succeed.
process_user_choice("1") :-
    do_something.
process_user_choice("2") :-
    do_something_else.
```

Process User Options

We create another predicate to process each option

Cuts might be necessary to avoid backtracking.

If `Choice == "0"` fails, do not run `process_choice/1` again.

```
process_user_choice("0"). % user choice is 0: succeed.
process_user_choice("1") :-
    do_something, !.
process_user_choice("2") :-
    do_something_else, !.
```

Process User Options

We create another predicate to process each option

We can also treat invalid options and force backtracking to the menu.

```
process_user_choice("1") :-
    do_something, !.
process_user_choice("2") :-
    do_something_else, !.
process_user_choice(_) :-
    writeln("Invalid option."),
    fail. % fail to return to the menu
```

Process User Options

We create another predicate to process each option

The exit case is just another case and other solutions are possible.

```
process_user_choice("0") :-
    writeln("See you next time!"), halt.
process_user_choice("1") :-
    do_something, !.
process_user_choice("2") :-
    do_something_else, !.
process_user_choice(_) :-
    writeln("Invalid option."),
    fail. % fail to return to the menu
```

Gathering Information

Requesting and saving information to dynamic predicate

The information provided by the user is stored in dynamic predicates.

```
:- dynamic info/1.
:- retractall(info(_)), asserta(info(_)).
```

Requesting and saving information to dynamic predicate

Note that we write an universal fact to the database.

```
:- dynamic info/1.
:- retractall(info(_)), asserta(info(_)).
```

That means that in the absence of user input, we assume `info/1` is true.

We do not eliminate any possible solution depending on `info`.

We can assume the opposite and suspend all conclusions in the absence of information. In that case, we do not call `asserta(info(_)).`

Gathering Information

Requesting and saving information to dynamic predicate

Before asking a question, we check whether the predicate is not already set.

```
:- dynamic info/1.
:- retractall(info(_)), asserta(info(_)).
ask_question :-
    info(X),
    var(X).
```

Gathering Information

Requesting and saving information to dynamic predicate

Another way to do it is to keep track of what we have already asked.

```
:- dynamic info/1.
:- retractall(info(_)), asserta(info(_)).
:- dynamic already_asked/1.

ask_question :-
    info(X),
    \+ already_asked(info).
```

Gathering Information

Requesting and saving information to dynamic predicate

Request the user's input and save the answer in the respective predicate.

```
:- dynamic info/1.
:- retractall(info(_)), asserta(info(_)).
ask_question :-
    info(X),
    var(X),
    writeln("Some options"),
    read_line_to_string(user_input, Choice),
    retractall(info(_)),
    assertz(info(Choice)).
```

Gathering Information

Requesting and saving information to dynamic predicate

Reading a string does not require the user to type a dot every time.

```
ask_question :-
    info(X),
    var(X),
    writeln("Some options"),
    read_line_to_string(user_input, Choice),
    retractall(info(_)),
    assertz(info(Choice)).
```

Requesting and saving information to dynamic predicate

If necessary, map `Choice` to the corresponding information.

```
ask_question :-
    info(X),
    var(X),
    writeln("Some options"),
    read_line_to_string(user_input, Choice),
    map(Choice, Info),
    retractall(info(_)),
    assertz(info(Info)).
```

Gathering Information

Requesting and saving information to dynamic predicate

Convert option numbers or letters to the corresponding information.

```
map("1", option1).
map("2", option2).
map("3", option3).
```

This also helps with treating invalid user choices.

`map/2` will fail it the choice is not listed.

Requesting and saving information to dynamic predicate

We can add a default case if we want to warn the user of an invalid choice.

```
map("1", option1) :- !.
map("2", option2) :- !.
map("3", option3) :- !.
map(_, _) :-
    writeln("Invalid option"),
    fail. % return to main menu.
```

Now we need cuts so we do not backtrack here if something fails later on.

Picking the next question

Implicit question order

We can write questions in an order that fits the use case.

```
ask_question :-
    infoA(A), var(A),
    ...
ask_question :-
    infoB(B), var(B),
    ...
ask_question :-
    infoN(N), var(N),
    ...
```

Picking the next question

Implicit question order

If `infoX` is yet unknown, `var(X)` is true and we pose the question.

Otherwise, we move on to the next one.

```
ask_question :-
    infoA(A), var(A),
    ...
ask_question :-
    infoB(B), var(B),
    ...
ask_question :-
    infoN(N), var(N),
    ...
```

Picking the next question

Explicit question order

We can also pick the next question given what is currently known.

```
ask_question :-
    infoA(A), A == optionA2,
    infoB(B), B == optionB3,
    infoX(X), var(X),
    ...
```

If we know infoA = optionA2 and infoB = optionB3, then ask about infoX.

Picking the next question

Explicit question order

We can also number the questions and have a predicate select the next one.

```
next_question(X),
ask_question(X),
...
next_question(4) :-
    infoA(A), A == optionA2,
    infoB(B), B == optionB3.
...
ask_question(4) :-
    infoD(D), var(D),
    ...
```

Inference

Forward and Backward Chaining

Two types of reasoning procedures.

Forward Chaining

We start from facts and apply rules to arrive at a conclusion.

Data-driven

Backward Chaining

We start with a hypothesis and look for facts and rules to prove it.

Goal-driven

Forward and Backward Chaining

Suppose we know that a fact A implies a fact B.

```
factA.
factB :- factA.
```

Forward Chaining

We know that A is true.

Therefore by the rule above B must also be true.

Backward Chaining

We want to prove B, our hypothesis.

We see that, by the rule above, B holds if A is true.

A is true, then we prove B.

Combination of forward and backward chaining

Expert systems are complex and usually call for chaining in both directions.

Diagnosis example

The patient mentions a first symptom.

forward: The doctor uses that to arrive at possible diagnoses (hypothesis).

backward: The doctor asks for other symptoms to confirm his hypotheses.

forward: With new symptoms, the doctor forms new hypothesis.

...

Combination of forward and backward chaining

Expert systems are complex and usually call for chaining in both directions.

User interaction

When interacting with the user, forward chaining is more common.

- Given the user information, provide an answer.

Backward chaining typically appears when selecting the next question.

- Form a hypothesis and ask a question to prove it.
- Select the most discriminative question.

Finding Solutions I

What can we infer with the information known?

Suppose we structure our knowledge base as follows.

```
% Knowledge base
hasproperty(infoA, solution1, optionA1).
hasproperty(infoB, solution2, optionB3).
...
% Properties model
property(infoA, optionA1).
property(infoA, optionA2).
...
```

Finding Solutions I

Inference rules

We can then write domain specific inference rules.

```
find_suggestions(Sug, Exp) :-
    infoA(A),
    infoB(B),
    suggestion_rule(A, B, Sug, Exp).
% domain specific inference rules
suggestion_rule(A, B, Sug, Exp) :-
    hasproperty(infoA, Sug, A),
    hasproperty(infoB, Sug, B),
    explain([A, B], Exp).
```

Explaining the solution

We can explain the suggestion by the gathering the relevant properties.

```
% Produces a list of pairs Property-Value.
explain([],[]).
explain([H|T],[Prop-H|T2]) :-
    property(Prop,H),
    explain(T,T2).
```

Finding Solutions I

Explaining the solution

The result is a list of pairs Property-Value that explains the solution.

```
% Knowledge base
infoA(optionA2).
infoB(optionB2).
solution(s1) :-
    infoA(optionA2), infoB(optionB2).

?- explain([optionA2, optionB2], Exp).
Exp = [infoA-optionA2, infoB-optionB2].
```

Finding Solutions II

What can we infer with the information known?

Suppose we have a knowledge base with a more general structure.

```
% Knowledge base
solution(s1) :-
    infoA(optionA1), infoB(optionB3), infoC(optionC2).
solution(s1) :-
    infoA(optionA2), infoB(optionB2).
solution(s2) :-
    infoA(optionA2), infoB(optionB4).
...
```

Finding Solutions II

Inference rules

Our suggestion rule only needs to find all viable solutions and pick one.

```
find_suggestions(Sug, Exp) :-
    infoA(A),
    infoB(B),
    suggestion_rule(A, B, Sug, Exp).
% domain specific inference rules
suggestion_rule(A, B, Sug, Exp) :-
    bagof(X, solution(X), Sugs), % fails if there is no solution.
    select_suggestion(Sugs, Sug),
    explain(Sug, Exp).
```

Finding Solutions II

Distinguishing possible solutions

For instance, we could pick the most popular whenever more than one option is available.

```
popularity(s1, 0.6).
popularity(s2, 0.4).
select_suggestion(Sugs, Sug) :-
    mostPopular(Sugs, Sug).
mostPopular(Sugs, Sug) :-
    findall(P, (member(S, Sugs), puplarity(S, P)), Pops),
    max_list(Pops, MaxPop),
    popularity(Sug, MaxPop).
```

Finding Solutions II

Explaining the solution

Built-in predicate `clause/2` provides the body of a rule.

```
explain(Sug, Exp) :-
    clause(solution(Sug), Exp).
```

With `clause/2` we obtain the list of goals that lead to the conclusion.

```
    (infoA(optionA1), infoB(optionB3)).
```

Ideally, we should parse this list into something the user can interpret.

```
    Solution s1 because you selected optionA1 and optionB3.
```

Breaking rules into goals

Built-in predicate `clause/2` provides the body of a rule.

```
solution(s1) :-
    infoA(optionA1), infoB(optionB3), infoC(optionC2).
solution(s1) :-
    infoA(optionA2), infoB(optionB2).

?- clause(solution(s1), Goals).
Goals = (infoA(optionA1), infoB(optionB3), infoC(optionC2));
Goals = (infoA(optionA2), infoB(optionB2)).
```

Breaking rules into goals

The second argument of `clause/2` is an actual goal we can query.

```
infoA(optionA2).
infoB(optionB2).
solution(s1) :-
    infoA(optionA1), infoB(optionB3), infoC(optionC2).
solution(s1) :-
    infoA(optionA2), infoB(optionB2).

?- clause(solution(s1), Goals), Goals.
Goals = (infoA(optionA2), infoB(optionB2)).
```

Breaking rules into goals

We can transform a set of goals into a list using the `=..` operator.

```
solution(s1) :-
    infoA(optionA1), infoB(optionB3), infoC(optionC2).
solution(s1) :-
    infoA(optionA2), infoB(optionB2).

?- clause(solution(s1), Goals), Goals=..List.
Goals = (infoA(optionA1), infoB(optionB3), infoC(optionC2)).
List = [',', infoA(optionA1), (infoB(optionB3), infoC(optionC2))].
```

Breaking rules into goals

Note how Prolog always represents rules as a conjunction of two goals.

```
Goals = (goal1, goal2, goal3, ..., goaln).
List = [',', goal1, (goal2, goal3, ..., goaln)].
```

Prove `goal1`, then prove `goal2, goal3, ..., goaln`

Prove `goal2`, then prove `goal3, ..., goaln`

...

Prove `goaln-1`, then prove `goaln`

Breaking rules into goals

Other useful built-in predicates `functor/3` and `arg/3`.

```
% Gets the name and arity of a predicate.
?- functor(my_predicate(A1, A2, A3), Name, Arity).
Name = my_predicate,
Arity = 3.

% Gets an argument of a predicate given its position.
?- arg(2, my_predicate(A1, A2, A3), Value).
A2 = Value.
?- arg(2, my_predicate(a1, a2, a3), Value).
Value = 2.
```

Breaking rules into goals

We can use break down goals into predicates and arguments.

```
Goals = (infoA(optionA1), infoB(optionB3), infoC(optionC2)).
?- functor(Goals, Name, Arity).
Name = (','),
Arity = 2.
?- arg(1, Goals, SubGoal1).
SubGoal1 = infoA(optionA1).
?- arg(2, Goals, SubGoal2).
SubGoal2 = (infoB(optionB3), infoC(optionC2)).
```

Breaking rules into goals

Given a set of goals, we can produce a list of reasons for a conclusion.

```
get_reasons(Goal, Reasons) :-
    functor(Goal, Pred, _), % Pred is the name of the predicate
    Pred \= ,, % Check if it is a conjunction of goals
    write_reason(Pred, Reasons), !.
% Split a conjunction into two subgoals
get_reasons(Goal, Reasons) :-
    arg(1, Goal, Subgoal_1), get_reasons(Subgoal_1, R1),
    arg(2, Goal, Subgoal_2), get_reasons(Subgoal_2, R2),
    append(R1, R2, Reasons).
```

Breaking rules into goals

Be careful for an unknown information `info(_)` is always true.

```
% For predicates with arity 1.
write_reason(Pred, [Value]) :-
    call(Pred, Value),
    nonvar(Value), !.
% If a variable is free, Value is an empty list.
write_reason(_, []).
```

Analysing Rules - Providing Explanations

Putting everything together

We call `clause/2` to get the body of the rule leading to the conclusion.

```
% Provide explanations
explain(Conclusion, Reasons):-
    clause(Conclusion, Rule),
    Rule, % get only rules that succeed
    !, % get only one rule
    get_reasons(Rule, Reasons).
```

Putting everything together

Then call `get_reasons` to break the body into a list of user options.

```
% Provide explanations
explain(Conclusion, Reasons):-
    clause(Conclusion, Rule),
    Rule, % get only rules that succeed
    !, % get only one rule
    get_reasons(Rule, Reasons).
```

Putting everything together

We get the list of options selected by the user that lead to the solution.

```
infoA(optionA2).
infoB(optionB2).
solution(s1) :-
    infoA(optionA1), infoB(optionB3), infoC(optionC2).
solution(s1) :-
    infoA(optionA2), infoB(optionB2).

?- explain(solution(s1), Reasons).
Reasons = [optionA2, optionB2].
```

Uncertainty and Probability

Probabilities as Degrees of Belief

Expressing uncertainty

A human expert might express his reasoning in terms of probabilities.

```
likes(X, redwine, 0.8) :-
    likes(X, coffee, 1.0).
```

If X likes coffee, then there is an 80% chance X will like red wine.

Expressing uncertainty

There can be uncertainties in both sides.

```
likes(X, redwine, P) :-
    likes(X, coffee, P2),
    P is P2 * 0.8.
```

If X likes coffee, then there is at most an 80% chance X will like red wine.

Probabilities as Degrees of Belief

Simple scheme for combining probabilities

We can manipulate probabilities with minimum and maximum operations.

```
% Given a list of probabilities,
% for the probability of observing all, take the minimum.
and(ListP, P) :-
    min_list(ListP, P).
% for the probability of observing any, take the maximum.
or(ListP, P) :-
    max_list(ListP, P).
```

That is a simplification and not a rigorous treatment of probabilities.

Combining the probabilities of observing multiple facts

If we want the probability of observing all, apply and.

If we want the probability of observing any, apply or.

```
likes(john, redwine, 0.7).
likes(john, coffee, 0.5).
likes_all(Person, List, Prob) :-
    findall(P, (member(Obj, List), likes(Person, Obj, P)), ListP),
    and(ListP, Prob).
likes_any(Person, List, Prob) :-
    findall(P, (member(Obj, List), likes(Person, Obj, P)), ListP),
    or(ListP, Prob).
```

Probabilities

Prolog does not support probabilities

A goal is either true or false, but we can still represent probabilities.

```
dice(1, 1/6). % (number, probability)
dice(2, 1/6).
dice(3, 1/6).
dice(4, 1/6).
dice(5, 1/6).
dice(6, 1/6).
```

Manipulating probabilities

Probability of a combination two numbers.

```
combProb(D1, D2, P) :-
    dice(D1, P1),
    dice(D2, P2),
    P is P1 * P2.
```

If we roll two dice, the probability of the outcome is the product of the probabilities of observing each individual number.

That is if we assume independent dice!

Manipulating probabilities

Probability of getting a given number as the sum of two dice.

```
% Get all possible combinations.
combDice(D, D1, D2) :-
    between(1, D, D1),
    D2 is D - D1.
% Get the probability of the sum.
probSum(S, Prob) :-
    findall(P,
        (combDice(S, D1, D2),
        combProb(D1, D2, P)), L),
    sum_list(L, Prob).
```

Manipulating probabilities

Independence assumption - probability of intersection and union of events

$$P(A \cap B) = P(A) * P(B)$$
$$P(A \cup B) = 1 - (1 - P(A)) * (1 - P(B))$$

```
indep_and([P], P).
indep_and([H|T], P) :-
    indep_and(T, PT), P is H*PT.

indep_or([P], P).
indep_or([H|T]], P) :-
    indep_or(T, PT), P is 1-((1-H)*(1-PT)).
```

Uncertainty of Rules

Probabilities in rules

Suppose we want to add probabilities to the following rule.

```
a :- b, c.
```

if b and c, then a.

Uncertainty of Rules

Probabilities in rules

Suppose now we want the probabilistic version of that rule.

```
a :- b, c.
```

if b and c, then a.

```
a(Pa) :- b(Pb), c(Pc),
    indep_and([Pb, Pc], Pa).
```

We can assume independence and calculate the probability of a.

Uncertainty of rules

The rule might not always apply or we might not be certain of its outcome.

```
a :- b, c.
```

if b and c, then a.

```
a(Pa) :- b(Pb), c(Pc),
    indep_and([Pb, Pc, 0.7], Pa).
```

We can model this uncertainty by adding another probability to the equation.

If b and c, then a 70% of the time.

Negation with Probabilities

Probabilities do not with negation

Suppose we have the following rule.

```
a :- b, \+ c.
```

if b and not c, then a.

Negation with Probabilities

Probabilities do not with negation

Suppose now we want to add probabilities to this reasoning.

```
a :- b, \+ c.
```

if b and not c, then a.

```
a(Pa) :- b(Pb), \+ c(Pc),
    indep_and([Pb, Pc], Pa).
```

Negation with Probabilities

Probabilities do not with negation

The previous approaches we have discussed do not apply to negation.

```
a :- b, \+ c.
```

if b and not c, then a.

```
a(Pa) :- b(Pb), \+ c(Pc),
    indep_and([Pb, Pc], Pa).
```

Any evidence for c will render a false!

Negation with Probabilities

One minus the probability of it being true

$$P(\neg X) = 1 - P(X)$$

```
a :- b, \+ c.
```

if b and not c, then a.

```
oneminus(P, P2) :- P2 is 1 - P.
a(Pa) :- b(Pb), c(Pc),
    oneminus(Pc, Pc2)
    indep_and([Pb, Pc2], Pa).
```

Example of an Expert System