Problem Solving and Search

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

[http://www.illc.uva.nl/~ulle/teaching/pss/]

Table of Contents

Lecture 1: Basic Prolog 3
Lecture 2: Working with Lists 29
Lecture 3: Working with Numbers 46
Lecture 4: Working with Operators 57
Lecture 5: Backtracking, Cuts and Negation 73
Lecture 6: Additional Features 96

Lecture 1: Basic Prolog

What is Prolog?

- Prolog is one of two *classical* programming languages developed specifically for applications in AI (the other one is Lisp).
- Prolog (*pro*gramming in *log*ic) is a *logic-based* programming language: programs correspond to sets of logical formulas and the Prolog interpreter uses logical methods to resolve queries.
- Prolog is a *declarative* language: you specify *what* problem you want to solve rather than *how* to solve it.
- Prolog often permits very *compact* solutions. So we can focus on underlying (mathematical) ideas rather than on software design.
- Prolog is very *useful* for classical problem solving tasks in AI (but less so for certain other tasks). Everyone should be familiar with multiple programming languages/paradigms!

Plan for Today

The objective of this first lecture is to introduce you to the most basic concepts of the Prolog programming language.

- Your first Prolog program: big and not so big animals
- Terminology: talking about Prolog programs
- How Prolog works: matching and query resolution

Facts

A little Prolog program consisting of four *facts:*

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

Queries

After compilation we can *query* the Prolog system:

```
?- bigger(donkey, dog).
Yes

?- bigger(monkey, elephant).
No
```

A Problem

The following query does not succeed!

```
?- bigger(elephant, monkey).
No
```

The *predicate* `bigger/2` apparently is not quite what we want.

What we really want: a predicate that succeeds whenever you can go from the first animal to the second by iterating the `bigger/2`-facts.

Mathematicians call this the *transitive closure* of `bigger/2`.

Rules

The following two *rules* define the new predicate `is_bigger/2` as the transitive closure of `bigger/2` (via *recursion*):

```
is_bigger(X, Y) :- bigger(X, Y).

is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

↑ “if”

↑ “and”

Now it works!

```
?- is_bigger(elephant, monkey).
Yes
```

Even better, we can use the *variable* `X`:

```
?- is_bigger(X, donkey).
X = horse ;
X = elephant ;
No
```

Press `;` (semicolon) to find alternative solutions. `No` at the end indicates that there are no further solutions.

Another Example

Are there any animals that are both smaller than a donkey and bigger than a monkey?

```
?- is_bigger(donkey, X), is_bigger(X, monkey).
No
```

Terms

We need some terminology to talk about the *syntax* of Prolog programs.

Prolog *terms* are either

- *numbers*,
- *atoms*,
- *variables*, or
- *compound terms*.

Atoms start with a lowercase letter or are enclosed in single quotes:

```
elephant, xYZ, a_123, 'How are you today?'
```

Variables start with a capital letter or the underscore:

```
X, Elephant, _G177, MyVariable, _
```

Terms (continued)

Compound terms consist of a *functor* (an atom) and a number of *arguments* (terms) enclosed in brackets:

```
is_bigger(horse, X)
f(g(Alpha, _), 7)
'My Functor'(dog)
```

Atoms and numbers are also called *atomic terms*.

Terms without variables are called *ground terms*.

Facts and Rules

Facts are compound terms (or atoms), followed by a full stop.
Facts are used to define something as being unconditionally true.

```
bigger(elephant, horse).
parent(john, mary).
```

Rules consist of a *head* and a *body* separated by `:-`. The head of a rule (a compound term or an atom) is true if all *goals* in the body (each a compound term or an atom) can be proved to be true.

```
grandfather(X, Y) :-
  father(X, Z),
  parent(Z, Y).
```

Facts and rules are used to define *predicates*.

Programs and Queries

Facts and rules are also called *clauses*. A *program* is a list of clauses.

You compose your programs in a text editor.

A *query* consists of one or several compound terms (or atoms), separated by commas and followed by a full stop.

You type your queries in at the Prolog prompt and expect a response.

```
?- is_bigger(horse, X), is_bigger(X, dog).
X = donkey
Yes
```

Built-in Predicates

Prolog comes with a large number of *built-in predicates*. Examples:

- Compiling a program file:

```
?- consult('big-animals.pl').
Yes
```

- Writing something (a term) on the screen:

```
?- write('Hello World!'), nl.
Hello World!
Yes
```

Matching

The concept of matching is at the core of Prolog's inner workings:

> Two terms *match* if they are identical or can be made identical by substituting their variables with suitable ground terms.

We can explicitly ask Prolog whether two given terms match by using the built-in equality-predicate = (written as an infix operator).

```
?- born(mary, yorkshire) = born(mary, X).
X = yorkshire
Yes
```

The variable instantiations are reported in Prolog's answer.

Examples

```
?- f(a, g(X, Y)) = f(X, Z), Z = g(W, h(X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes

?- p(X, 2, 2) = p(1, Y, X).
No
```

The Anonymous Variable

The variable _ (underscore) is called the *anonymous variable*.

Every occurrence of _ represents a different variable (which is why instantiations are not being reported).

```
?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes
```

Answering Queries

Answering a query means proving that the goal represented by that query can be satisfied (given the program currently in memory).

Recall: Programs are lists of facts and rules.

- A fact declares something as being true.
- A rule states conditions for something being true.

Answering Queries (continued)

To answer a query, Prolog executes this algorithm:

- If a goal matches with a *fact*, then it is satisfied.
- If a goal matches the *head of a rule*, then it is satisfied if the goal represented by the rule's body is satisfied.
- If a goal consists of several *subgoals* separated by commas, then it is satisfied if all its subgoals are satisfied.
- When trying to satisfy goals using *built-in predicates* (such as `write/1`), Prolog also performs the action associated with it (such as writing something on the screen).

Example: Mortal Philosophers

Consider the following argument:

All men are mortal.
Socrates is a man.

Hence, Socrates is mortal.

It has two *premises* and a *conclusion*.

Translating it into Prolog

The two premises can be expressed as a little Prolog program:

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

The conclusion can then be formulated as a query:

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

Goal Execution

(1) The query `mortal(socrates)` is made the initial goal.

(2) Prolog looks for the first matching fact or head of rule and finds `mortal(X)`. Variable instantiation: `X = socrates`.

(3) This variable instantiation is extended to the rule's body, so `man(X)` becomes `man(socrates)`.

(4) New goal: `man(socrates)`.

(5) Success, because `man(socrates)` is a fact itself.

(6) Therefore, also the initial goal succeeds.

Programming Style

It is extremely important that you write programs that are easily understood by others! Some guidelines:

- Use *comments* to explain what you are doing:

```
/* This is a long comment, stretching over several
lines, which explains in detail how I have implemented
the aunt/2 predicate ... */

aunt(X, Z) :-
  sister(X, Y),  % This is a short comment.
  parent(Y, Z).
```

Programming Style (continued)

- Separate clauses by one or more blank lines.
- Write only one predicate per line and use indentation:

```
blond(X) :-
  father(Father, X),
  blond(Father),
  mother(Mother, X),
  blond(Mother).
```

  (Very short clauses may also be written in a single line.)

- Insert a space after every comma inside a compound term:

```
born(mary, yorkshire, '01/01/1999')
```

- Write short clauses with bodies consisting of only a few goals.
  If necessary, split into shorter sub-clauses.
- Choose meaningful names for your variables and atoms.

Summary: Terminology

- All Prolog expressions are made up of *terms* (which are either numbers, atoms, variables, or compound terms).
- *Atoms* start with lowercase letters or are enclosed in single quotes. *Variables* start with capital letters or the underscore.
- Prolog programs are lists of *facts* and *rules* (a.k.a. *clauses*).
- *Queries* are submitted to the system to initiate a computation.
- Some *built-in predicates* have special meaning.

Summary: Answering Queries

- When answering a user's query, Prolog tries to prove that the corresponding goal can be satisfied (can be made true).

  This is done using the rules and facts given in a program.

- The current goal is *matched* with the first possible fact or rule head. In the latter case, the rule's body becomes the new goal.

- The *variable instantiations* made during matching are carried along throughout the computation and reported at the end.

- Only the *anonymous variable* _ can be instantiated differently whenever it occurs.

Lecture 2: Working with Lists

Plan for Today

One of the most useful data structures in Prolog are *lists*. The aim of this lecture is to show you how lists are represented in Prolog and to introduce you to the basic principles of working with lists.

Lists in Prolog

An example for a Prolog list:

```
[elephant, horse, donkey, dog]
```

Lists are enclosed in *square brackets*. Their elements could be any Prolog terms (including other lists). The *empty list* is `[]`.

Another example:

```
[a, X, [], f(X,y), 47, [a,b,c], bigger(cow,dog)]
```

Internal Representation

Internally, the list

```
[a, b, c]
```

is represented by the term

```
.(a, .(b, .(c, [])))
```

That means, this is *just a new notation*. Internally, lists are just compound terms with the functor `.` (dot) and the special atom `[]` as an argument on the innermost level.

We can verify this also within Prolog:

```
?- X = .(a, .(b, .(c, []))).
X = [a, b, c]
Yes
```

Remark: Recent versions of SWI-Prolog use `'[|]'` instead of `.` (dot).

The Bar Notation

If a *bar* `|` is put just before the last term in a list, this means that this last term denotes a sub-list. Inserting the elements before the bar at the beginning of the sub-list yields the entire list.

For example, `[a, b, c, d]` is the same as `[a, b | [c, d]]`.

Examples

Extract the second element from a given list:

```
?- [a, b, c, d, e] = [_, X | _].
X = b
Yes
```

Make sure the first element is a 1 and get the sub-list after the second:

```
?- MyList = [1, 2, 3, 4, 5], MyList = [1, _ | Rest].
MyList = [1, 2, 3, 4, 5]
Rest = [3, 4, 5]
Yes
```

Head and Tail

The first element of a list is called its *head*. The rest of the list is called its *tail*. (The empty list does not have a head.)

A special case of the bar notation—with exactly one element before the bar—is called the *head/tail-pattern*. It can be used to extract head and/or tail from a list. Example:

```
?- [elephant, horse, tiger, dog] = [Head | Tail].
Head = elephant
Tail = [horse, tiger, dog]
Yes
```

Head and Tail (continued)

Another example:

```
?- [elephant] = [X | Y].
X = elephant
Y = []
Yes
```

Note: The tail of a list is always a list itself. The head of a list is an element of that list. In principle, the head can itself be a list as well (but it typically is not).

Appending Lists

We now want to write a predicate `concat_lists/3` to concatenate (append) two given lists.

It should work like this:

```
?- concat_lists([1, 2, 3, 4], [dog, cow, tiger], L).
L = [1, 2, 3, 4, dog, cow, tiger]
Yes
```

Solution

The predicate `concat_lists/3` is implemented *recursively*.

The *base case* applies when the first list happens to be empty.

In every *recursion step* we take off the head and use the same predicate again, with the (shorter) tail, until we reach the base case.

```
concat_lists([], List, List).

concat_lists([Elem|List1], List2, [Elem|List3]) :-
  concat_lists(List1, List2, List3).
```

We can do more!

We can also use `concat_lists/3` to decompose a given list:

```
?- concat_lists(Begin, End, [1, 2, 3]).
Begin = []
End = [1, 2, 3] ;
Begin = [1]
End = [2, 3] ;
Begin = [1, 2]
End = [3] ;
Begin = [1, 2, 3]
End = [] ;
No
```

Built-in Predicates for List Manipulation

`append/3`: Append two lists (same as our `concat_lists/3`).

```
?- append([1, 2, 3], List, [1, 2, 3, 4, 5]).
List = [4, 5]
Yes
```

`length/2`: Get the length of a list.

```
?- length([tiger, donkey, cow, tiger], N).
N = 4
Yes
```

Membership

`member/2`: Test for list membership.

```
?- member(tiger, [dog, tiger, elephant, horse]).
Yes
```

Backtracking into `member/2`:

```
?- member(X, [dog, tiger, elephant]).
X = dog ;
X = tiger ;
X = elephant ;
No
```

Example

Consider the following program:

```
show(List) :-
  member(Element, List),
  write(Element),
  nl,
  fail.
```

Note: `fail/0` is a built-in predicate that always fails.

What happens when you submit a query such as the following one?

```
?- show([elephant, horse, donkey, dog]).
```

Example (continued)

```
?- show([elephant, horse, donkey, dog]).
elephant
horse
donkey
dog
No
```

The `fail` at the end of the rule causes Prolog to backtrack.
The subgoal `member(Element, List)` is the only choicepoint.
In every backtracking cycle a new element of `List` is matched with the variable `Element`. Eventually, the query fails (`No`).

More Built-in Predicates

`reverse/2`: Reverse the order of elements in a list.

```
?- reverse([1, 2, 3, 4, 5], X).
X = [5, 4, 3, 2, 1]
Yes
```

More built-in predicates can be found in the (online) reference manual.

Summary: Working with Lists

- List notation:
  - normal: `[Elem1, Elem2, Elem3]` (empty list: `[]`)
  - internal: `.(Elem1, .(Elem2, .(Elem3, [])))`
  - bar notation: `[Elem1, Elem2 | Rest]`
  - head/tail-pattern: `[Head | Tail]`
- Many predicates can be implemented recursively, exploiting the head/tail-pattern. (This is a central concept in Prolog!)
- Built-in predicates: `append/3`, `member/2`, `length/2`, ...

Lecture 3: Working with Numbers

Plan for Today

Prolog comes with a range of predefined arithmetic functions and operators. Expressions such as `3 + 5` are valid Prolog terms.

So, what's happening here?

```
?- 3 + 5 = 8.
No
```

The objective of this lecture is to clarify this (supposed) problem and to explain how to work with *arithmetic expressions* in Prolog.

Matching vs. Arithmetic Evaluation

The terms `3 + 5` and `8` *do not match*. In fact, if we are interested in the sum of the numbers 3 and 5, we cannot get it through matching, but we need to use *arithmetic evaluation*.

We have to use the `is`-*operator:*

```
?- X is 3 + 5.
X = 8
Yes
```

The `is`-Operator

The `is`-operator works as follows:

> *Evaluate* the term to its right as an arithmetic expression and then *match* the resulting number with the term to its left.

So the lefthand term should usually (basically: always) be a variable.

Example:

```
?- Value is 3 * 4 + 5 * 6, OtherValue is Value / 11.
Value = 42
OtherValue = 3.818181818181183
Yes
```

Note the small rounding error above.

A Subtle Detail

Beware that different Prolog systems may deal differently with the following kind of example:

```
?- X is 3.5 + 4.5.
X = 8
Yes
```

```
?- X is 3.5 + 4.5
X = 8.0
Yes
```

Some systems will try to instantiate `X` with an *integer* such as `8` whenever possible; some will instantiate `X` with a *float* such as `8.0`.

That is, in the second case the following query would fail:

```
?- X is 3.5 + 4.5, X = 8.
```

Example: Length of a List

Instead of using `length/2` we can now write our own predicate to compute the length of a list:

```
len([], 0).

len([_ | Tail], N) :-
  len(Tail, N1),
  N is N1 + 1.
```

Functions

Prolog provides a number of built-in *arithmetic functions* that can be used with the `is`-operator. See reference manual for details.

Examples:

```
?- X is max(8, 6) - sqrt(2.25) * 2.
X = 5.0
Yes

?- X is (47 mod 7) ** 3.
X = 125
Yes
```

Relations

Arithmetic relations are used to compare two arithmetic values.

Example:

```
?- 2 * 3 > sqrt(30).
Yes
```

The following relations are available:

`=:=`	arithmetic equality	`=\=`	arithmetic inequality
`>`	greater than	`>=`	greater than or equal
`<`	less than	`=<`	less than or equal

Examples

Recall the difference between *matching* and *arithmetic evaluation:*

```
?- 3 + 5 = 5 + 3.
No

?- 3 + 5 =:= 5 + 3.
Yes
```

Recall the *operator precedence* of arithmetics:

```
?- 2 + 3 * 4 =:= (2 + 3) * 4.
No
?- 2 + 3 * 4 =:= 2 + (3 * 4).
Yes
```

Programming Style

To check whether 8 equals 3 plus 5, this works, but is extremely ugly:

```
?- 8 is 3 + 5.
Yes
```

It works, because evaluating the term `3 + 5` arithmetically yields the number `8`, which indeed matches the term on the left.

It is ugly, because, semantically, what you are trying to do here is to compare the values of two arithmetic expressions, not evaluate one.

So you should use an arithmetic relation:

```
?- 8 =:= 3 + 5.
Yes
```

Summary: Working with Numbers

- For logical pattern matching continue to use the predicate =, but for *arithmetic evaluation* use the `is`-operator.
- A range of built-in *arithmetic functions* is available (some of them are written as infix operators, such as +).
- Arithmetic expressions can be compared using *arithmetic relations* such as < or =:= (i.e., not using the `is`-operator).

Lecture 4: Working with Operators

Plan for Today

Operators provide a more convenient way of writing certain expressions in Prolog that could otherwise be difficult to read for humans.

For example, we can write `3 * 155` instead of `*(3, 155)`, and we can write `N is M + 1` instead of `is(N, +(M, 1))`.

Both forms of notation are considered equivalent. So matching works:

```
?- +(1000, 1) = 1000 + 1.
Yes
```

The main objective of this lecture is to show you how you can define your own *operators* in Prolog. In the process, you will learn a few things about how computers interpret the *structure* of an expression.

Operator Precedence

Some operators bind stronger than others. In mathematics, for example, * (multiplication) binds stronger than + (addition).
The degree to which an operator is binding is called its *precedence*.

In Prolog, operator precedences are encoded by means of numbers (in SWI-Prolog between 0 and 1200). The arithmetic operator *, for example, has precedence 400; + has precedence 500. Thus

the lower an operator's precedence value, the stronger it binds

This is why Prolog is able to compute the correct result for the following example (i.e., not 25):

```
?- X is 2 + 3 * 5.
X = 17
Yes
```

Precedence of Terms

The precedence of a term is the precedence of its *principal operator*.

If the principal functor is not (written as) an operator or if the term is enclosed in brackets, then the precedence value is defined as 0.

Examples:

- The precedence of `3 + 5` is 500.
- The precedence of `3 * 3 + 5 * 5` is also 500.
- The precedence of `sqrt(3 + 5)` is 0.
- The precedence of `elephant` is 0.
- The precedence of `(3 + 5)` is 0.
- The precedence of `3 * +(5, 6)` is 400.

Operator Types

Operators can be divided into three groups:

- *infix operators*, like + in Prolog
- *prefix operators*, like – for negative numbers
- *postfix operators*, like ! in mathematics (factorial)

Is fixing the type of an operator and its precedence enough for Prolog to fully "understand" the structure of a term using that operator?

Example

Consider the following example:

```
?- X is 25 - 10 - 3.
X = 12
Yes
```

Why not 18?

So, clearly, precedence and type alone are *not* enough to fully specify the structural properties of an operator.

Operator Associativity

We also have to specify the *associativity* of an operator: e.g., `-` is left-associative. So `25 - 10 - 3` is interpreted as `(25 - 10) - 3`.

In Prolog, associativity is represented by atoms such as `yfx`: `f` indicates the position of the operator (i.e., `yfx` denotes an infix operator) and `x` and `y` indicate the positions of the arguments.

A `y` should be read as:

> at this position a term with a precedence *less than or equal* to that of the operator has to occur

But `x` means that:

> at this position a term with a precedence *strictly less* than that of the operator has to occur

Understand how this makes the interpretation of `25 - 10 - 3` unambiguous (note that `-` is defined using the pattern `yfx`)!

Associativity Patterns

Pattern	Associativity		Examples
`yfx`	infix	left-associative	`+, -, *`
`xfy`	infix	right-associative	`,` (for subgoals)
`xfx`	infix	non-associative	`=, is, <` (i.e., no nesting)
`yfy`	makes no sense, structuring would be impossible		
`fy`	prefix	associative	
`fx`	prefix	non-associative	
`yf`	postfix	associative	
`xf`	postfix	non-associative	

Checking Precedence and Associativity

You can use the built-in predicate `current_op/3` to check precedence and associativity of currently defined operators.

```
?- current_op(Prec, Assoc, *).
Prec = 400
Assoc = yfx
Yes

?- current_op(Prec, Assoc, is).
Prec = 700
Assoc = xfx
Yes
```

Overloading of Operator Names

The same operator symbol can be used once as a binary and once as a unary operator. Example:

```
?- current_op(Prec, Assoc, -).

Prec = 200
Assoc = fy ;

Prec = 500
Assoc = yfx ;

No
```

Defining Operators

New operators are defined using the op/3-predicate, submitting the operator definition as a query. Terms using the new operator will then be equivalent to terms using the operator as a normal functor.

Example:

```
?- op(400, xfx, is_bigger).
Yes

?- is_bigger(dog, cat) = dog is_bigger cat.
Yes.
```

Assuming our big-animal program has been compiled, this will work:

```
?- elephant is_bigger dog.
Yes
```

Aside: Query Execution at Compilation Time

You can add queries to a program file (using `:-` as a prefix operator). They are executed whenever the program is compiled.

Suppose the file `my-file.pl` contains this line:

```
:- write('Hello, have a beautiful day!').
```

This will have the following effect:

```
?- consult('my-file.pl').
Hello, have a beautiful day!
my-file.pl compiled, 0.00 sec, 224 bytes.
Yes
?-
```

Operator Definition at Compilation Time

You can do the same for operator definitions. For example, the line

```
:- op(200, fy, small).
```

inside a program file will cause a prefix operator called `small` to be declared whenever the file is compiled. It can then be used inside the program itself, in other programs, and in user queries.

Term Decomposition

Recall that a *compound term* consists of a *functor* and one or more *arguments*. (An *atomic term* has no arguments.)

Given a term `T`, the predicate `=../2` (defined as an infix operator) can be used to generate a list, the head of which is the functor of `T` and the tail of which is the list of arguments of `T`:

```
?- loves(john,mary) =.. List.
List = [loves, john, mary]
Yes

?- elephant =.. List.
List = [elephant]
Yes

?- 5 + 3 =.. List.
List = [+, 5, 3].
Yes
```

Composing Terms

You can also use =../2 to compose new terms:

```
?- member(X, [f,g,h]), Y =.. [X,a,b].
X = f
Y = f(a, b) ;
X = g
Y = g(a, b) ;
X = h
Y = h(a, b) ;
No
```

This is very useful, because using a variable in the position of a functor would cause a syntax error (for most Prolog systems):

```
?- member(X, [f,g,h]), Y = X(a,b).
ERROR: Syntax error: Operator expected
```

Summary: Working with Operators

- The structural properties of an operator are determined by its *precedence* (a number) and its *associativity pattern* (e.g., `yfx`).
- Use `current_op/3` to check operator definitions.
- Use `op/3` to make your own operator definitions.
- Operator definitions are usually included inside a program file as queries (using `:-`, i.e., like a rule without a head).
- The built-in predicate `=../2` can be used to *de/compose terms*. It is declared as a (non-associative) infix operator.

Lecture 5: Backtracking, Cuts and Negation

Plan for Today

In this lecture, we are going to look in more detail into how Prolog evaluates queries, in particular into the process of *backtracking*.

We are going to discuss both the benefits of backtracking and some of the problems it creates, and see how to control backtracking (via *cuts*).

We are also going to discuss the closely related subject of *negation*.

Backtracking

Subgoals that can be satisfied in more than one way are *choicepoints*.

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

This is an example for a choicepoint, because the variable `X` could be matched with either `a`, `b`, or `c`.

During goal execution Prolog keeps track of choicepoints. If one path turns out to be a failure, it jumps back to the most recent choicepoint and tries the next alternative. This is known as *backtracking*.

Smart Use of Backtracking

Given a list in the first argument position, `permutation/2` generates all possible permutations of that list in the second argument through *enforced backtracking* (if the user presses `;` after every solution):

```
permutation([], []).

permutation(List, [Element | Permutation]) :-
  select(Element, List, Rest),
  permutation(Rest, Permutation).
```

Recall that `select/3` checks whether the element in the first argument position can be matched with an element of the list in the second argument position; if so, the term in the third argument position is matched with the remainder of that list.

Example

```
?- permutation([1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
No
```

Problems with Backtracking

Asking for alternative solutions generates wrong answers for this first attempt at implementing a predicate to remove duplicates from a list:

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
  member(Head, Tail),
  remove_duplicates(Tail, Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
  remove_duplicates(Tail, Result).
```

Example

```
?- remove_duplicates([a, b, b, c, a], List).
List = [b, c, a] ;
List = [b, b, c, a] ;
List = [a, b, c, a] ;
List = [a, b, b, c, a] ;
No
```

Do you recognise the pattern of what goes wrong?

Introducing Cuts

Sometimes we want to prevent Prolog from backtracking into certain choicepoints, to either *eliminate wrong solutions* or *improve efficiency*.

This is possible by using a *cut*, written as `!`. This built-in predicate always succeeds and prevents Prolog from backtracking into subgoals placed before the cut inside the same rule body.

Example

The correct program for removing duplicates from a list:

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
  member(Head, Tail), !,
  remove_duplicates(Tail, Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
  remove_duplicates(Tail, Result).
```

Cuts

When executing the subgoals in a rule's body the so-called *parent goal* is the goal that caused the matching of the head of the current rule.

Definition of the functionality of a cut:

> Whenever a cut is encountered in a rule's body, all choices made between the time that rule's head has been matched with the parent goal and the time the cut is passed are final, i.e., any choicepoints are being discarded.

Exercise

Using cuts (but without using negation), implement a predicate `add/3` to add an element to a list, unless that element already is a member of the list. Make sure there are no wrong alternative solutions.

Examples:

```
?- add(elephant, [dog, donkey, rabbit], List).
List = [elephant, dog, donkey, rabbit] ;
No

?- add(donkey, [dog, donkey, rabbit], List).
List = [dog, donkey, rabbit] ;
No
```

Solution

```
add(Element, List, List) :-
  member(Element, List), !.

add(Element, List, [Element | List]).
```

Problems with Cuts

The predicate add/3 does not work as expected when the last argument is already instantiated! Example:

```
?- add(dog, [dog, cat, bird], [dog, dog, cat, bird]).
Yes
```

We could use the following implementation of add/3 instead:

```
add(Element, List, Result) :-
  member(Element, List), !,
  Result = List.

add(Element, List, [Element | List]).
```

While this solves the problem, it also emphasises that using cuts can be tricky and affects the declarative character of Prolog ...

Summary: Backtracking and Cuts

- *Backtracking* is the mechanism by which Prolog can find all alternative solutions to a given query.
- So: Prolog provides the search strategy, not the programmer! This is why Prolog is called a *declarative* language.
- Carefully placed *cuts* (`!`) can be used to prevent Prolog from backtracking into certain subgoals. This may make a program more efficient and/or avoid (wrong) alternative answers.
- But: Cuts destroy the declarative character of a Prolog program (which, for instance, makes finding mistakes a lot harder).

  So use them sparingly!

Example

Consider the following Prolog program:

```
animal(elephant).
animal(donkey).
animal(tiger).
```

And now observe the system's reaction to the following queries:

```
?- animal(donkey).
Yes

?- animal(duckbill).
No
```

Wrong answer? Why?

The Closed World Assumption

In Prolog, `Yes` means the statement in question is *provably true*.
So `No` just means the statement is *not provably true*.

This is *not* the same as to say that the statement is *false*. We can only infer that the statement is false if we are willing to assume that *all relevant information* is present in our Prolog program.

For the semantics of Prolog programs we usually make this so-called *Closed World Assumption:* we assume that nothing outside of the world described by a given Prolog program exists (is true).

The \+-Operator

If what we want to know is not whether a given goal succeeds, but rather whether it fails, we can use the *\+-operator* (*negation*).

The goal `\+ Goal` succeeds if and only if `Goal` fails. Example:

```
?- \+ member(17, [1, 2, 3, 4, 5]).
Yes
```

This is known as *negation as failure:* Prolog's negation is defined via its failure to provide a proof.

Negation as Failure: Example

Consider the following program:

```
married(peter, lucy).
married(paul, mary).
married(bob, juliet).
married(harry, geraldine).

single(Person) :-
  \+ married(Person, _),
  \+ married(_, Person).
```

Example (continued)

After compilation, Prolog reacts as follows (recall that Mary is married to Paul, while our little database does not mention Claudia at all):

```
?- single(mary).
No

?- single(claudia).
Yes
```

In the *closed world* described by our Prolog program, Claudia has to be single, because she is not known to be married.

Where to use \+

Note that the \+-operator can only be used to negate *goals*. These are either (sub)goals in the *body of a rule* or (sub)goals of a *query*.

We cannot negate facts or the heads of rules, because this would actually constitute a redefinition of the \+-operator (in other words: an explicit definition of Prolog's negation, which would not be compatible with the closed world assumption).

Connection: Cuts and Negation as Failure

We can implement our own version of the \+-operator by using a cut:

```
neg(Goal) :- call(Goal), !, fail.
neg(_).
```

Recall that `fail/0` is a built-in predicate that always fails.

The built-in predicate `call/1` takes a goal as argument and executes it.

Examples:

```
?- neg(member(17, [1, 2, 3, 4, 5])).
Yes
```

```
?- neg(member(4, [1, 2, 3, 4, 5])).
No
```

Disjunction

We have seen *conjunction* ("and", comma) and *negation* ("not", \+).

We actually also know about *disjunction* ("or") already, given that several rules with the same head amount to a disjunction.

But disjunction can also be implemented directly *within* one rule by using the operator ; (semicolon). Example:

```
parent(X, Y) :- father(X, Y); mother(X, Y).
```

This is equivalent to the following program:

```
parent(X, Y) :- father(X, Y).

parent(X, Y) :- mother(X, Y).
```

Use the disjunction operator sparingly. It tends to make programs harder to read.

Summary: Negation and Disjunction

- *Closed World Assumption:* In Prolog everything that cannot be proven from the given facts and rules is considered false.
- *Negation as Failure:* Prolog's negation operator is implemented as the failure to provide a proof for a given statement.
- Goals can be negated using the *\+-operator*.

  Always use \+, not the `not`-operator, as the latter may mean different things in different Prolog systems.
- A *disjunction* of goals can be expressed by using ; (semicolon). (The comma between two subgoals denotes a conjunction.)

Lecture 6: Additional Features

Plan for Today

The purpose of this lecture is to introduce some additional features, going beyond the core of the Prolog language we have covered so far.

- Collecting answers: `findall/3` etc.
- Dynamic predicates: `assert/1` and `retract/1`
- Input/output and file handling

The first are *control features* (just like cuts and negation as failure), while the other two are often useful for *larger programming projects* (important, but not the focus of this course).

Backtracking and Alternative Answers

Next we are going to see how to collect all alternative answers to a given query (or goal) in a list.

Assume the following program has been consulted:

```
student(ann, 44711, pass).
student(bob, 50815, pass).
student(pat, 41018, fail).
student(sue, 41704, pass).
```

We can get all the answers to a query by forcing Prolog to backtrack:

```
?- student(Name, _, pass).
Name = ann ;
Name = bob ;
Name = sue ;
No
```

Collecting Answers in a List

Instead, the `findall/3` predicate can be used to collect these answers in a single list. Examples:

```
?- findall(Name, student(Name,_,pass), List).
List = [ann, bob, sue]
Yes

?- findall(Name, student(Name,_,dunno), List).
List = []
Yes
```

Specification of findall/3

Schema: `findall(+Template, +Goal, -List)`

Prolog will search for all possible solutions to the `Goal` (backtracking). For every solution found, the necessary instantiations to `Template` are made, and these instantiations are collected in the list `List`.

So `Template` and `Goal` should (usually) share one or more variables. Variables occurring in `Goal` but not in `Template` can have any value (these are not being reported).

Another Example

Here is again our program:

```
student(ann, 44711, pass).
student(bob, 50815, pass).
student(pat, 41018, fail).
student(sue, 41704, pass).
```

An example with a complex goal and a template with two variables:

```
?- Goal = (student(Name,ID,Grade), ID < 50000),
   findall(Name/Grade, Goal, List).
Goal = (student(Name, ID, Grade), ID<50000),
List = [ann/pass, pat/fail, sue/pass]
Yes
```

Collecting Answers with bagof/3

The `bagof/3` predicate is similar to `findall/3`, but now the values taken by variables occurring in the goal but not the template do matter and a different list is created for every possible instantiation of these variables. Example:

```
?- bagof(Name/ID, student(Name,ID,Grade), List).
Grade = fail
List = [pat/41018] ;

Grade = pass
List = [ann/44711, bob/50815, sue/41704] ;
No
```

Example with an Unbound Variable

In the following query we say that we are not interested in the value of `ID` (by using the `^` operator), but Prolog will still give alternative solutions for every possible instantiation of `Grade`:

```
?- bagof(Name, ID^student(Name,ID,Grade), List).
Grade = fail
List = [pat] ;

Grade = pass
List = [ann, bob, sue] ;
No
```

Summary: Collecting Answers

- `findall/3` collects all the answers to a given *goal* that match a given *template* in a *list*. Variables not occurring in the template may take different values within the list of answers.
- `bagof/3` is similar, but generates a different list for every possible instantiation of the variables not occurring in the template. Use the `Var^` construct to allow for a variable to take different values within the same list of answers.
- `setof/3` works like `bagof/3`, but will also remove any duplicates and return lists that are ordered.
- Note that `findall/3` returns an empty list if the goal in the second argument position cannot be satisfied, while `bagof/3` and `setof/3` will simply fail. (Observe that this makes sense!)
- Use these predicates sparingly! They tend to tempt people into writing inelegant and inefficient programs.

Assert and Retract

Prolog evaluates any queries with respect to a given knowledge base (your program + definitions of built-in predicates).

It is possible to *dynamically* add clauses to this knowledge base:

- Executing a goal of the form `assert(+Clause)` will add the clause `Clause` to the Prolog knowledge base.
- Executing `retract(+Clause)` will remove that clause again.
- Executing `retractall(+Clause)` will remove *all* the clauses matching `Clause`.

A typical application is to dynamically create and change a database. In that case, the `Clauses` usually are facts.

Database Example

```
?- assert(zoo(monkey)), assert(zoo(camel)).
Yes

?- zoo(X).
X = monkey ;
X = camel ;
No

?- retract(zoo(monkey)).
Yes

?- zoo(X).
X = camel ;
No
```

Dynamic Manipulation of the Program

You can even declare your program predicates as being dynamic and assert and retract clauses for these predicates.

Example: Suppose we have consulted our "big animals" program from the first lecture (see next slide for a reminder) and suppose we have declared `bigger/2` as a dynamic predicate ...

```
?- is_bigger(camel, monkey).
No

?- assert(bigger(camel,horse)).
Yes

?- is_bigger(camel, monkey).
Yes
```

The Big Animals Program

```
:- dynamic bigger/2.

bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Fast Fibonacci

Another application of dynamic predicates would be to store previously computed answers, rather than to compute them again from scratch every time they are needed. Example:

```
:- dynamic fibo/2.
fibo(0, 0). fibo(1, 1).
fibo(N, F) :-
  N >= 2, N1 is N - 1, N2 is N - 2,
  fibo(N1, F1), fibo(N2, F2), F is F1 + F2,
  asserta(fibo(N,F):-!). % assert as first clause
```

This is much faster than the standard program for the Fibonacci sequence (= the above program without the last line). However, a solution with just a single recursive call is even better than this one.

Summary: Dynamic Predicates

- Use `assert/1` to add clauses to the knowledge base and use `retract/1` or `retractall/1` to remove them again.
- If the predicate to be asserted or retracted is already in use, then it needs to be declared as being dynamic first (in SWI-Prolog use the `dynamic` directive; this might differ for other systems).
- If the order of clauses in the dynamic knowledge base matters, there are further predicates such as `asserta/1` and `assertz/1` that can be used (check the reference manual for details).
- Note: Use with care! Dynamic predicates obfuscate the declarative meaning of Prolog and make it much harder to check programs (the same code will behave differently for different dynamic knowledge bases). Often a sign of poor programming style!

Input/Output and File Handling

We have already seen how to explicitly *write output* onto the user's terminal using the `write/1` predicate. Example:

```
?- X is 5 * 7, write('The result is: '), write(X).
The result is: 35
X = 35
Yes
```

Now we are also going to see how to *read input*. In Prolog, input and output from and to the user are similar to input and output from and to *files*, so we deal with these in one go.

Streams

In Prolog, input and output happen with respect to two *streams:* the current input stream and the current output stream. Each of these streams could be either the user's terminal (default) or a file.

You can *choose* the current *output stream* to be `Stream` by executing the goal `tell(+Stream)`. Example:

```
?- tell('example.txt').
Yes
```

Now `write/1` will not write to the user's terminal anymore, but instead to the file `example.txt`:

```
?- write('Hello, have a beautiful day!').
Yes
```

To *close* the current output stream, use the command `told/0`.

Reading Terms from the Input Stream

The corresponding predicates for choosing and closing an *input stream* are called `see/1` and `seen/0`.

To read from the current input stream, use the predicate `read/1`.

Note: Reading only works if the input stream is a sequence of *terms*, each of which is followed by a full stop (like in a Prolog program file).

Example: Reading Terms from a File

```
?- see('students.txt').
Yes
?- read(Next).
Next = student(ann, 44711, pass)
Yes
?- read(Next).
Next = student(bob, 50815, pass)
Yes
?- read(Next).
Next = student(pat, 41018, fail)
Yes
?- read(Next).
Next = end_of_file
Yes
?- seen.
Yes
```

Content of file `students.txt`:

```
% Database of students
student(ann, 44711, pass).
student(bob, 50815, pass).
student(pat, 41018, fail).
```

Example with User Input

Consider the following program:

```
start :-
  write('Enter a number followed by a full stop: '),
  read(Number),
  Square is Number * Number,
  write('Square: '),
  write(Square).
```

After compilation, it works as follows:

```
?- start.
Enter a number followed by a full stop: 17.
Square: 289
Yes
```

Summary: Input/Output and File Handling

- Input and output use the concept of streams, with the default input/output stream being the user's terminal.
- Main predicates:
  - `see/1`: choose an input stream
  - `seen/0`: close the current input stream
  - `tell/1`: choose an output stream
  - `told/0`: close the current output stream
  - `read/1`: read the next term from the input stream
  - `write/1`: write to the output stream
- Using `read/1` only works with text files that are sequences of Prolog terms. To work with arbitrary files, have a look at `get/1`.
- There are many more input/output predicates in the manual.