

Rule-Based Systems: Logic Programming

Logic Programming

- Logic programming is the use of
 - ◆ logic as a declarative representation language
 - ◆ Backward chaining as inference rule
- Logic Programming is the basis of the programming language PROLOG

Logic Programs – A Sequence of Horn Clauses

- The sentences of a logic program are Horn clauses
 - Facts: H
 - Rules: $H \leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_n$
- A Horn clause without any head H is called a query
 - ◆ Query: $\leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_n$
- Queries are not part of a logic program, they start the inference

Knowledge Bases in PROLOG

- PROLOG (= PROgramming in LOGic) is a programming language based on Horn clauses
- A Prolog knowledge base consists of **clauses**.
- Clauses are either **facts** or **rules**. Each clause ends with a period "."
- Facts consist of just one literal:

```
is_a_dog(pluto) .  
is_a_sailor(popeye) .  
eats(popeye, spinach) .
```

- Rules consist of a **head** and a **body**

```
is_strong(popeye) :- eats(popeye, spinach) .  
is_strong_sailor(popeye) :- is_a_sailor(popeye) ,  
                             eats(popeye, spinach) .
```

- ◆ Head and body are separated by the symbol ":-" which means „ \leftarrow “
- ◆ Literals in the body of a rule are separated by comma „**,**“
(the comma is equivalent to the logical AND or „ \wedge “)
- ◆ The second rule can be read as: **is_strong_sailor(popeye)** is true of
is_a_sailor(popeye) is true and **eats(popeye, spinach)** is true.

Predicates and Literals

- Predicates are the building blocks of clauses
- Predicates have a name and arguments (parameters). Arity is the number of arguments.
- Predicates combine values which “make sense” together (are true)
- Examples:
 - ◆ `person(peter)`
 - ◆ `married(peter, cindy)`
 - ◆ `appointment(knut, “AB1”, “Lecture KE”)`
 - ◆ `not female(knut)`
- Literals are predicates and negated predicates

Predicate Expressions (Literals)

- Clauses contain literals (predicate expressions), which consist of a **predicate symbol** and arguments. The arguments are called **terms**, e.g.

`eats` (`popeye`, `spinach`) .

`eats` is a predicate symbol,
`popeye` and `spinach` are terms

- Predicates can have arbitrary arity. Arity is the number of arguments:

`it_rains` .

`it_rains`/0 has arity 0.

`eats` (`popeye`, `spinach`) .

`eats`/2 has arity 2.

`eats_spinach` (`popeye`) .

`eats_spinach`/1 has arity 1.

- The facts `friends` (`popeye`, `pluto`, `garfield`) and `friends` (`pluto`, `mickey`) define two different predicates, namely `friends`/3 (arity 3) and `friends`/2 (arity 2).

- The predicate expressions `eats` (`popeye`, `spinach`) and `eats_spinach` (`popeye`) express the same statements but are not equivalent

- The order of the arguments is significant: `father` (`john`, `paul`) is not the same as `father` (`paul`, `john`) . We determine which argument position should stand for what, but then we have to keep it:

Terms

- The basic data structure in Prolog are **terms**. They are arguments of predicates.
- Terms are either **simple** or **compound**.
- Simple terms in Prolog are **constants** and **variables**
- The constants are **symbols** and **numbers**.
- Compound terms are either **complex terms** or **lists**.

Simple Terms: Atoms, Numbers, Variables

Atoms are strings that begin with lowercase letters and consist only of letters, numbers, and the underscore, or strings that are enclosed in quotation marks:

`popeye, dog13XYZ, my_dog, "Lea?! @", 'Homer Simpson'`

Numbers are integers or floats:

`123, 89.5, 0, -323`

Variables are strings that begin with a capital letter or an underscore and consist only of letters, numbers, and the underscore:

`X, Variable, _x, _123, Hund_123, _`

Hints:

- ♦ Terms should always be 'speaking'.
- ♦ The `_` variable, which consists only of the underscore, is the anonymous variable.

Variables

```
exists(X) .  
  
likes(Everybody, mickey) -  
  
is_strong(X) :- has_trained(X) .  
friend(X,Y) :- friend(Y,X) .
```

- Variables can be used in facts, rules and queries.
- Same variables stand for the same values
- The clause `exists(X) .` is a universal fact (fact with an open variable).

Complex Terms

- Compound terms consist of a **functor** and any number of **arguments**.
 - ◆ The functor is always an atom.
 - ◆ The arguments are simple or complex terms.
- Functors – like predicate symbols – have an arity (number of arguments).
- Examples of complex terms:

loves (**mary**, **father** (**mary**))

friends (**x**, **father** (**father** (**popeye**)))

"Mary loves her father"
- **Note:** In the second example, **father/1** is a function symbol and not a predicate symbol

Queries about Facts

```
is_a_dog(pluto) .  
is_a_dog(snoopy) .  
is_a_sailor(popeye) .  
eats(popeye, spinach) .
```

- Inference in Prolog starts with a query. The system concludes whether the statement is true.
- Requests are made to the interpreter in the console and evaluated.
- A query about facts just checks whether the literal is in the knowledge base:

```
?- is_a_dog(pluto) .  
.  
  
?- eats(popeye, spinach) .
```

Queries with Variables

```
is_a_sailor(popeye) .  
eats(popeye,spinach) .  
likes(pluto,mickey) .  
likes(mickey,pluto) .  
likes(minnie,mickey) .  
likes(mickey,minnie) .
```

- If a query contains variables, the interpreter tries to instantiate the variable (i.e. assign a value) in such a way that the statement becomes true
- The assignment of the variables is displayed as a response
- By entering the semicolon (or clicking "Next") the interpreter looks for more answers

```
?- eats(popeye,X) .
```

```
?- likes(X,Y) .
```

Exercises

- Write as a logic program
 - ♦ john is a person
 - ♦ peter and mary are persons
 - ♦ fhnw is a university
 - ♦ john is matriculated at fhnw
 - ♦ A student is a person who is matriculated at a university.
- ♦ Queries
 - ♦ Is john a student?

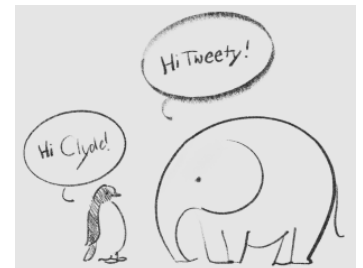
Exercise

- Represent in Prolog
 - Peter is the Father of Mary
 - Peter is the Father of John
 - Mary is the Mother of Mark
 - Jane is the Mother of Jane
- Write rules for
 - grandfather
 - grandmother
 - sibling (father or mother in common)

Predicates

- All clauses, whose head have the same predicate symbol and same arity define a predicate.

```
living_being(X) :- mammal(X) .  
living_being(X) :- bird(X) .  
  
mammal(X) :- elephant(X) .  
mammal(X) :- cat(X) .  
  
bird(X)      :- penguin(X) .  
  
elephant(clyde) .  
  
penguin(tweety) .
```



Exercises

- Write as a logic program
 - ◆ knut is a person
 - ◆ "KEBI" is a course
 - ◆ courses belong to study programs
 - ◆ bachelor and master are study programs
 - ◆ computer science is a master program
 - ◆ "KEBI" belongs to computer science
 - ◆ courses are taught by teachers
 - ◆ john attends course "KEBI"
 - ◆ students are attending courses
 - ◆ a student who attends a course of a master program is a master student
 - ◆ knut teaches "KEBI"
- ◆ Queries
 - ◆ Is john a master student?
 - ◆ Is knut a teacher?

Inference Procedure

Reasoning in Prolog

- Prolog's principle of automatic reasoning is based on
 - ◆ the principle of **unification** and
 - ◆ **backward chaining** with backtracking.
- To prove a target clause, Prolog tries to **unify** the clause with the facts and rule heads given in the knowledge base.
- If the query contains variables, a valid variable assignment (substitution) must be found.

Inference Procedure for Logic Programming

Let *resolvent* be the query $?- Q_1, \dots, Q_m$

While *resolvent* is not empty **do**

1. **Choose** a query literal Q_i from *resolvent*.
2. **Choose** a renamed¹ clause $H :- B_1, \dots, B_n$ from P such that Q_i and H unify with an most general **unifier** σ , i.e. $Q_i\sigma = H\sigma$ (Head Unification)
3. **If** no such Q_i and clause exist, then **backtrack**
4. Remove Q_i from the *resolvent*
5. **Add** B_1, \dots, B_n to the *resolvent*
6. Add σ to σ_{all}
7. Apply substitution σ to the *resolvent* and go to 1.

If *resolvent* is empty, **return** σ_{all} , else **return failure**.

¹ Renaming means that the variables in the clause get new unique identifiers



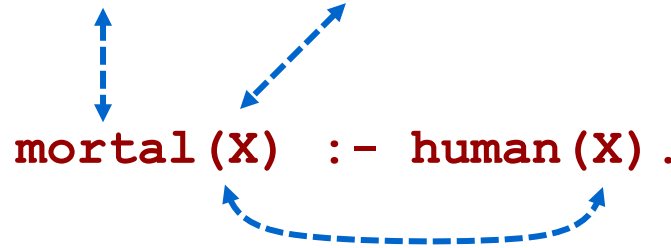
Unification

Head Unification

- A predicate from a query must be unifiable with the head of a clause.

- ♦ Query: `?- mortal(socrates) .`

- ♦ Clause: `mortal(X) :- human(X) .`



- Head Unification

- ♦ predicate symbols are equal
- ♦ Substitution: **$X = \text{socrates}$**

- New query: `?- human(socrates) .`

Unification

- Two expressions Q and H unify if there exists a substitution σ for any variables in the expressions so that the expressions are made identical ($Q\sigma = H\sigma$)

Unification Rules

- A constant unifies only with itself
- Two structures unify if and only if
 - ◆ they have the same (function or) predicate symbol and the same number of arguments, and
 - ◆ the corresponding arguments unify recursively
- An unbound variable unifies with anything

Substitution

- A *substitution* is a finite set of the form $\sigma = \{v_1 / t_1, \dots, v_n / t_n\}$
 - ♦ v_i 's: distinct variables.
 - ♦ t_i 's: terms with $t_i \neq v_i$.
- Applying a substitution σ to an expression E means to replace each occurrence of a variables v_i with the value t_i

■ Example:

$E = p(X, Y, f(a))$
$\sigma = \{X / b, Y / Z\}$
$E\sigma = p(b, Z, f(a))$

$E = \text{father}(\text{peter}, X)$
$\sigma = \{X / \text{mary}\}$
$E\sigma = \text{father}(\text{peter}, \text{mary})$

Unifier

- A substitution σ is a *unifier* of expressions E and F iff

$$E\sigma = F\sigma$$

- Example: Let E and F be two expressions:

- ♦ $E = f(x, b, g(z))$,
- ♦ $F = f(f(y), y, g(u))$.

Then $\sigma = \{x / f(b), y / b, z / u\}$ is a unifier of E and F :

- ♦ $E\sigma = f(f(b), b, g(u))$,
- ♦ $F\sigma = f(f(b), b, g(u))$

- A unifier σ of E and F is *most general* iff is more general than any other unifier of E and F , i.e. for any other unifier ρ there exists a unifier τ such that $\rho = \tau \circ \sigma$

Unification

- In unification, two terms are compared with each other or checked whether they can be equated (unified) by a suitable variable assignment.
- Unification is a part of reasoning. However, there is also the built-in predicate `=`, which equates two terms.

Unification Rule

Two terms are unifiable if and only if

- they are equal, or
- there is a substitution that assigns values to the variables in such a way that the two terms become equal

Unification of Terms

- If one of the terms is a variable, then the variable can be substituted with the other term

```
?- friend(popeye) = X.
```

- If a variable occurs more than once in a term, the variable assignment must be compatible everywhere

```
?- X=Y, X=popeye.
```

```
?- X=popeye, X=pluto.
```

Unification of Complex Terms

- Complex terms match exactly when:
 - 1) the terms have the same functor and the same arity, **and**
 - 2) match all corresponding arguments match, **and**
 - 3) the variable assignments are compatible with each other.

```
?- food(bread,X) = food(Y,sausage) .
```

```
?- meal(food(bread), drink(beer)) = meal(X,drink(Y)) .
```

```
?- food(bread,X,beer) = food(Y,sausage,X)
```

Backward Chaining

Rules and Inferences

- If the rule body is true (i.e. can be derived from the knowledge base), then the rule head is also true.
- This principle of deduction is called Modus Ponens:

$a \rightarrow b$	$b :- a.$	<code>is_strong(popeye) :- eats(popeye, spinach) .</code>
a	$a.$	<code>eats(popeye, spinach) .</code>
<hr/>	<hr/>	<hr/>
b	$b.$	<code>is_strong(popeye) .</code>

- From the rule `is_strong(popeye) :- eats(popeye, spinach) .` and the fact `eats(popeye, spinach) .` the Prolog interpreter infers that `is_strong(popeye) .` applies.

Rules and Queries

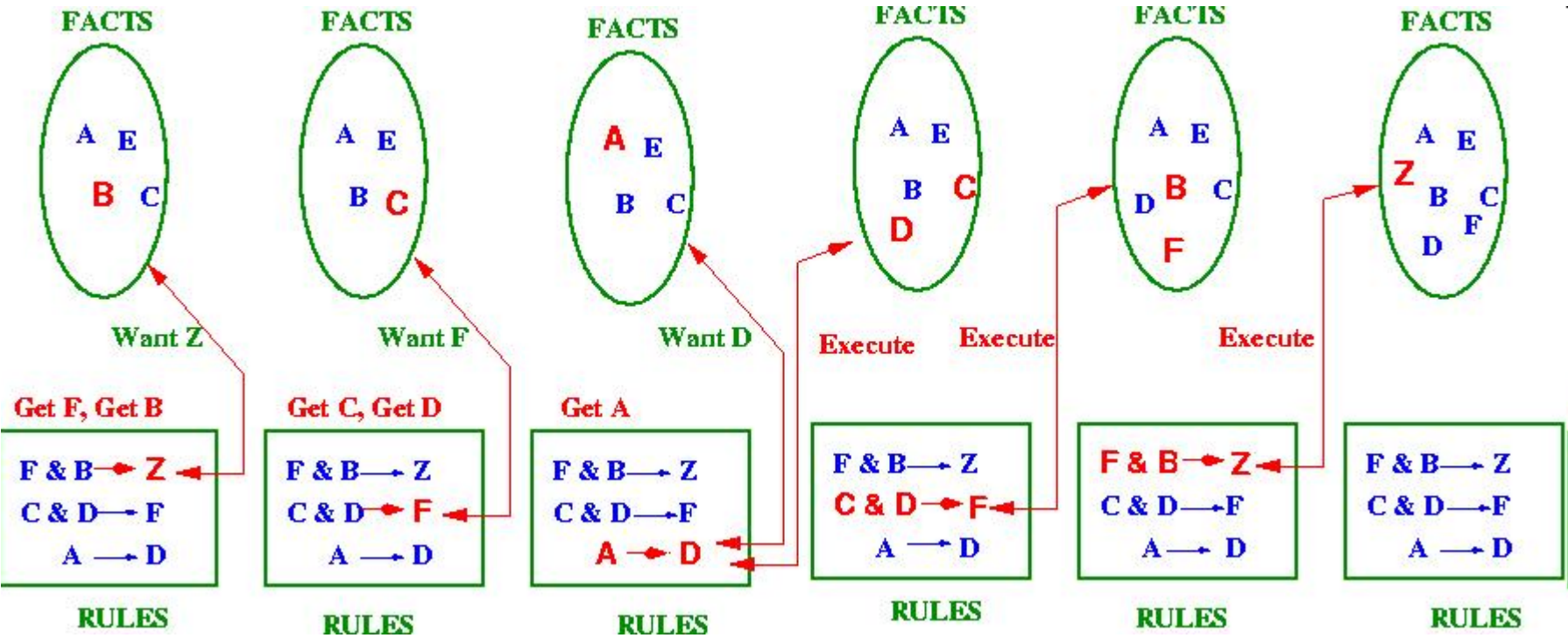
```
is_a_sailor(popeye) .  
eats(popeye, spinach) .  
  
is_strong(popeye) :- eats(popeye, spinach) .  
  
has_muscles(popeye) :- has_trained(popeye) .  
has_muscles(popeye) :- is_strong(popeye) .
```

- To answer queries, the rules are applied backwards.
 - ◆ If the query matches a fact, the query is true
 - ◆ If the query matches the header of a rule, the body becomes the new query

```
is_strong(popeye) :- eats(popeye, spinach) .  
?- is_strong(popeye) .
```

```
eats(popeye, spinach) .  
?- eats(popeye, spinach) .
```

Illustrating Backward Chaining



Source: Kerber (2004), <http://www.cs.bham.ac.uk/~mmk/Teaching/AI/I2.html>

Two Choices in the Inference Procedure

There are two choices in the inference procedure where a decision needs to be made

■ **Step 1: Selecting the Literal Q_i from the Resolvents**

- ◆ Solution in Logical Programming: **left-most goal**

■ **Step 2: Choosing a clause**

- ◆ Solution in logical programming: **top-most clause**
 - The clauses are selected in the order in which they appear.
 - **Backtracking**: If a selected clause does not succeed and there are alternative clauses, the next one is selected.

Within a Rule: From left to right

- Rule bodies and queries are proven from left to right.
- Only when a proof of the literal i in a rule is found, then literal $i+1$ can be proven

- Example Query:

```
?- female(X) , sibling(X,Y) .
```

- Example Rule

```
sister(X,Y) :- female(X) , sibling(X,Y) .
```

- First `female(X)` , is proven and then `sibling(X,Y)`

Choosing a Clause: Top-Down

- Head Unification is performed top down.
- The interpreter searches the database from top to bottom to find suitable clauses for proof

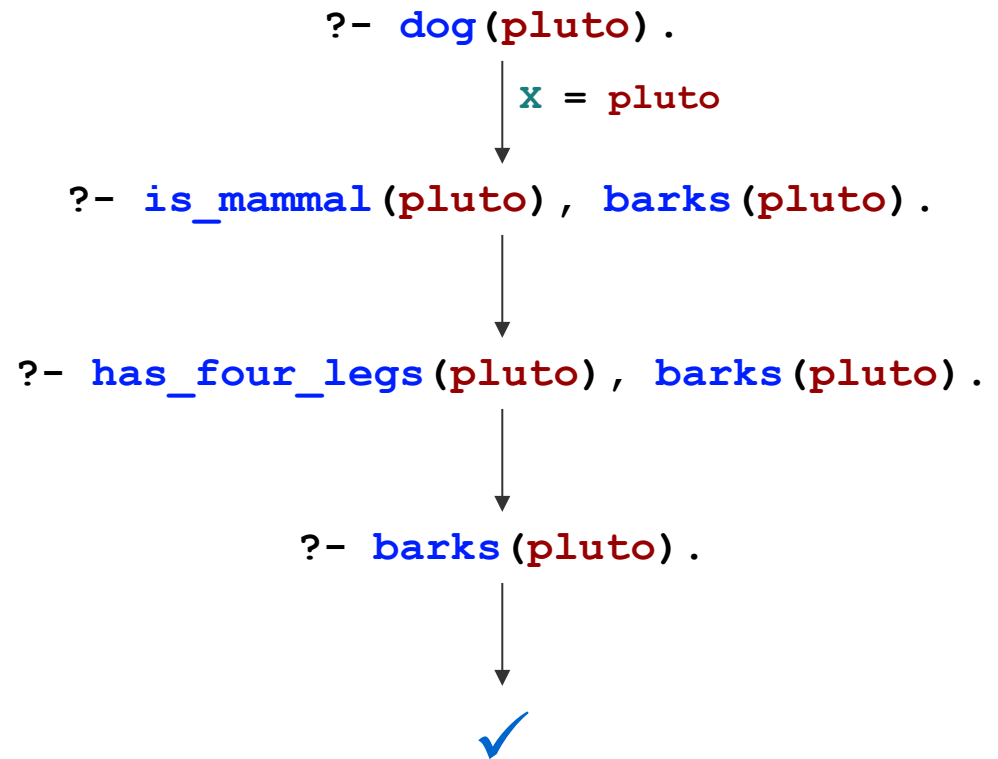
```
eat_spinach(popeye) .  
has_trained(garfield) .  
is_strong(X) :- has_trained(X) .  
is_strong(X) :- eat_spinach(X) .
```

- What is the first answer the the query:

```
?- is_strong(X) .
```

Search Tree for Depth-First Search

```
is_honest(pluto) .  
has_four_legs(pluto) .  
barks(pluto) .  
  
dog(X) :-  
    is_mammal(X) ,  
    barks(X) .  
  
is_mammal(X) :-  
    has_four_legs(X) .  
  
?- dog(pluto) .
```



Search Tree for Depth-First Search

```
is_honest(pluto) .
has_four_legs(pluto) .
barks(pluto) .

dog(X) :-
    is_mammal(X) ,
    barks(X) .

is_mammal(X) :-
    has_four_legs(X) .

?- dog(pluto) .
```

```
[trace] 8 ?- trace, dog(pluto) .
Call: (7) dog(pluto) ?
Call: (8) is_mammal(pluto) ?
Call: (9) has_four_legs(pluto) ?
Exit: (9) has_four_legs(pluto) ?
Exit: (8) is_mammal(pluto) ?
Call: (8) barks(pluto) ?
Exit: (8) barks(pluto) ?
Exit: (7) dog(pluto) ?
true.
```

A Logic Program and Queries

```
father (peter, mary) .
father (peter, john) .

mother (mary, mark) .
mother (jane, mary) .
```

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

grandmother (X, Z) :- mother (X, Y) , father (Y, Z) .
grandmother (X, Z) :- mother (X, Y) , mother (Y, Z) .

sibling (Y, Z) :- father (X, Y) , father (X, Z) .
sibling (Y, Z) :- mother (X, Y) , mother (X, Z) .
```

Queries :

```
?- father (peter, john) .
?- father (peter, X) .
?- grandfather (peter, mark) .
?- grandfather (peter, mary) .
?- grandfather (peter, S) .
?- sibling (X, Y) .
```

Adding Goal to Resolvent

- In step 5 of the Inference procedure the literals of the clause are added to the resolvent.
- Depending on whether the literals are added at the beginning or the end of the resolvent, we get two different strategies:
 - ◆ Adding the literals to the beginning of the resolvent gives **depth-first search**.
 - ◆ Adding the literals to the end of the resolvent gives **breadth-first search**.



Backtracking

Backtracking: Depth-First Search

- Backtracking can be triggered by two causes:
 - ◆ There is no further clause for the current query predicate.
 - ◆ An alternative solution is to be calculated.
- In any case, the interpreter goes back to the last branch in the proof tree, where alternatives were still open (depth-first).

```
eat_spinach(popeye) .  
has_trained(garfield) .  
is_strong(X) :- has_trained(X) .  
is_strong(X) :- eat_spinach(X) .
```

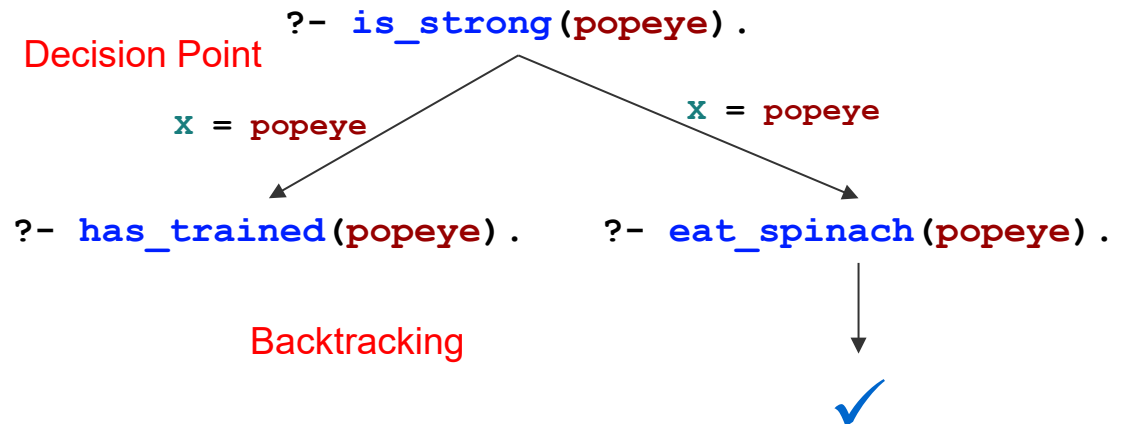
```
?- is_strong(popeye) .  
?- is_strong(X) .
```

Source: Wiebke Petersen, Grundkurs Prolog, HHU Düsseldorf, https://user.phil.hhu.de/~petersen/WiSe2324_Prolog/WiSe2324_Prolog.html

Search Tree: Decision Point and Backtracking

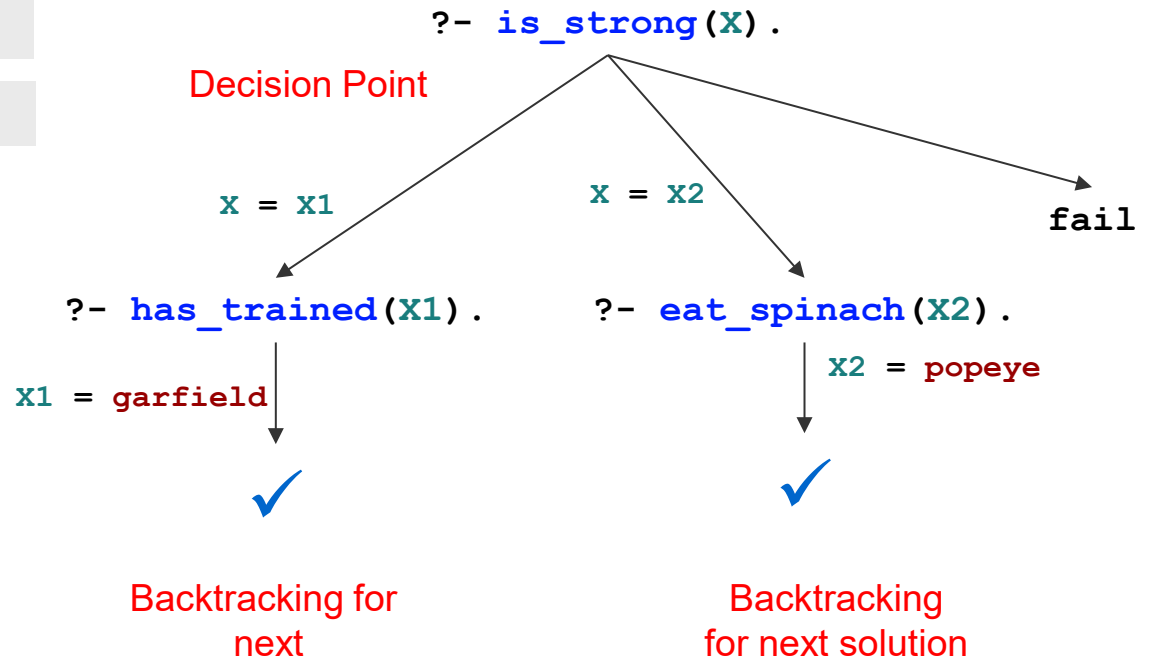
```
eat_spinach(popeye) .  
has_trained(garfield) .  
is_strong(X) :- has_trained(X) .  
is_strong(X) :- eat_spinach(X) .
```

```
?- is_strong(popeye) .
```

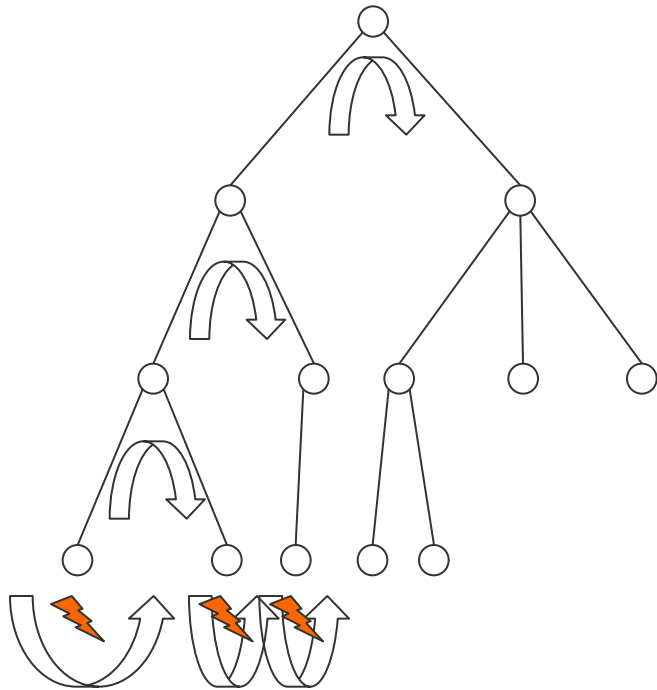


```
eat_spinach(popeye) .  
has_trained(garfield) .  
is_strong(X) :- has_trained(X) .  
is_strong(X) :- eat_spinach(X) .
```

```
?- is_strong(X) .
```



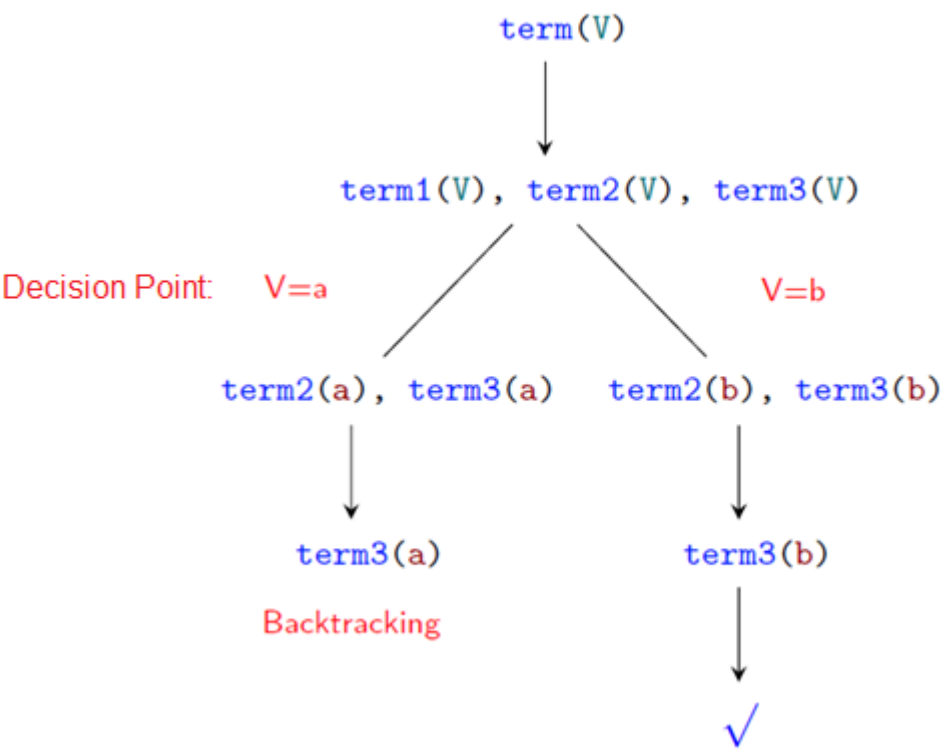
Backtracking



- Record any decision (**choose**) and its alternative
- If backtracking, then go back to the last decision and try another option
- When backtracking then roll back to the former situation (esp. for *resolvent* and σ_{all})

Search Tree: Decision Points

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



Source: Wiebke Petersen, Grundkurs Prolog, HHU Düsseldorf, https://user.phil.hhu.de/~petersen/WiSe2324_Prolog/WiSe2324_Prolog.html

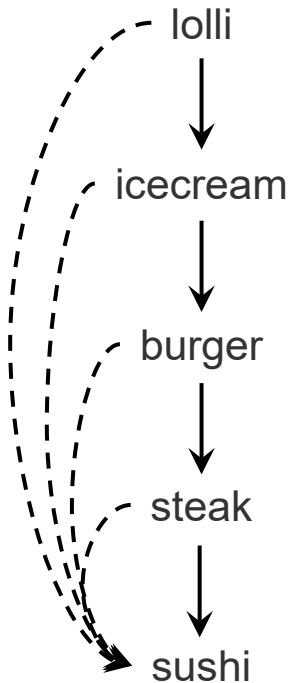
Prolog 's Solution: Summary

- Choice of a query literal:
→ **leftmost** literal first
- Choice of a clause
→ **Topmost clause first - with backtracking**
- Adding new goal to the resolvent
→ **At the beginning.**



Recursion

Task



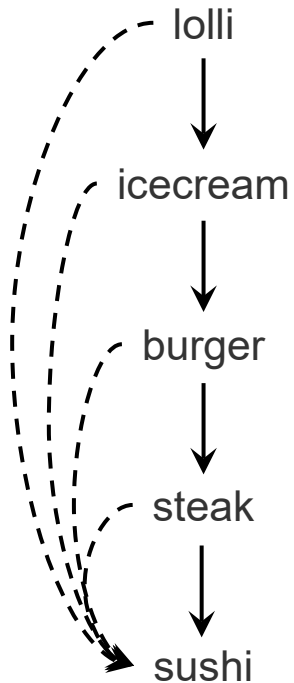
costs_less: →
cheaper: -.->

- Given these facts:

```
costs_less(lolli,icecream) .  
costs_less(icecream,burger) .  
costs_less(burger,steak) .  
costs_less(steak,sushi) .
```

Write rules for cheaper/2. such that cheaper(X,Y) is true, if X costs less than Y.

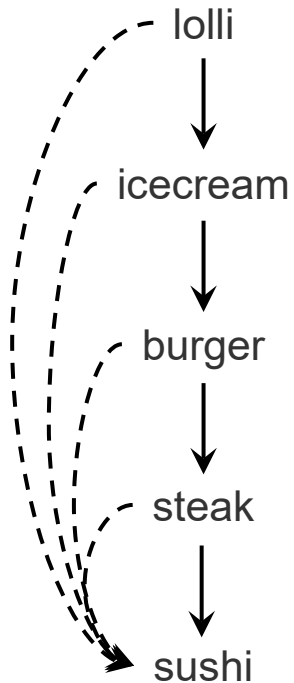
Nicht-rekursive Definition



costs_less: →
cheaper: -.->

```
costs_less(lolli,icecream) .  
costs_less(icecream,burger) .  
costs_less(burger,steak) .  
costs_less(steak,sushi) .
```


Solution: Recursive Predicate



costs_less: →
cheaper: -.->

```
costs_less(lolli,icecream) .  
costs_less(icecream,burger) .  
costs_less(burger,steak) .  
costs_less(steak,sushi) .
```

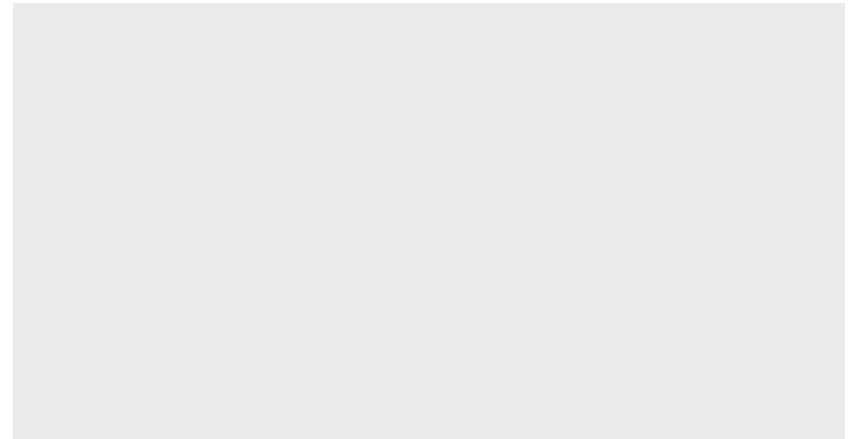
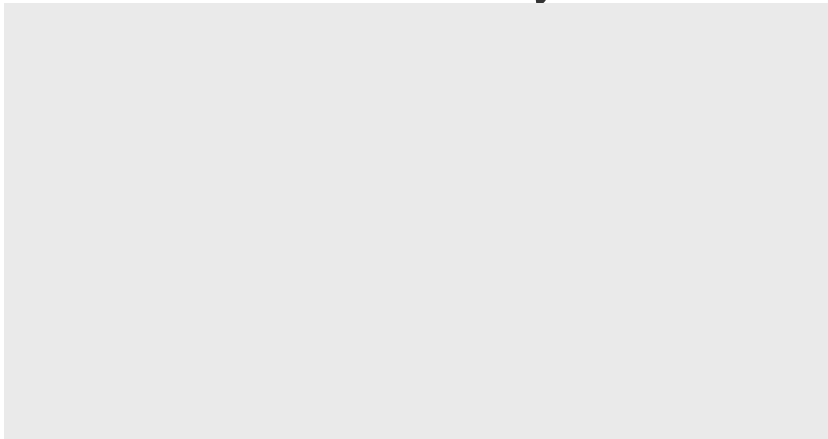
Recursion

```
likes(pluto, mickey) .  
likes(mickey, pluto) .  
likes(minnie, mickey) .  
likes(mickey, minnie) .
```

- In the knowledge base we see that pluto likes mickey and als mickey likes pluto. The same for minnie and mickey.
- Assume that likes is a inverse predicate. How can we avoid to write all facts. Assume want to write only one likes fact for a couple and get the inverse by inference.

Recursive Predicates

- A predicate is defined recursively when the predicate in the rule head is called in one of the defining rules.
- The basic idea is to reduce a common task to a simpler task of the same class (loops).
- Recursion makes it possible to write compact predicate definitions and avoid redundancy.



Declarative und procedural Interpretation of a Knowledge Base

Declarative Interpretation

- Declarative meaning is the meaning that is 'meant' or 'expressed' when reading the knowledge base as a set of logical statements.
- The declarative meaning of a prolog program can be defined as the set of all statements that can be logically derived from the knowledge base

Procedural Interpretation

- Procedural interpretation is the meaning that comes from what Prolog 'does' with a knowledge base.
- The procedural meaning of a Prolog program can be defined as the set of all queries (statements) for which the Prolog interpreter finds a variable assignment that results in the output **true**

Procedural Interpretation of Recursive Predicates

```
cheaper (X, Y) :- costs_less (X, Y) .
cheaper (X, Y) :- costs_less (X, Z) ,
                  cheaper (Z, Y) .
```

- First rule: To prove that X is cheaper than Y, it is enough to prove that X costs less than Y.
- Second rule: To prove that X is cheaper than Y, this problem can be broken down into two sub-problems. We are looking for a Z so that X costs less than Z (subproblem 1) and that Z is cheaper than Y (subproblem 2).

Procedural und declarative Interpretation

- As a reminder, Prolog works its way
 - ♦ through the knowledge base from top to bottom,
 - ♦ within the clauses from left to right.
- How does the order affect the procedural behavior of the predicate?

```
parent(john,peter) .  
parent(mary,john) .  
parent(susan,mary) .
```

```
ancestor1(X,Y) :- parent(X,Y) .  
ancestor1(X,Z) :- parent(X,Y) ,  
                    ancestor1(Y,Z) .
```

```
ancestor2(X,Z) :- parent(X,Y) ,  
                    ancestor2(Y,Z) .  
ancestor2(X,Y) :- parent(X,Y) .
```

```
ancestor3(X,Y) :- parent(X,Y) .  
ancestor3(X,Z) :- ancestor3(Y,Z) ,  
                    parent(X,Y) .
```

```
ancestor4(X,Z) :- ancestor4(Y,Z) ,  
                    parent(X,Y) .  
ancestor4(X,Y) :- parent(X,Y) .
```

Quelle: Wiebke Petersen, Grundkurs Prolog, HHU Düsseldorf, https://user.phil.hhu.de/~petersen/WiSe2324_Prolog/WiSe2324_Prolog.html

Defining "harmless" recursive Predicates

- Recursive predicates always require at least two clauses:
 - ◆ a recursive clause
 - ◆ an anchor or exit clause.
- The anchor clause should always precede the recursive clause (otherwise there is a risk of an infinite loop).
- In the rule body of the recursive clause, it often makes sense to put the recursive call at the end.

```
cheaper(X,Y) :-  
    costs_less(X,Y) .  
cheaper(X,Y) :-  
    costs_less(X,Z) ,  
    cheaper(Z,Y) .
```

anchor clause

Objectives: Recursive predicates,

- that do not lead to endless loops,
- who schedule as early as possible,
- which can be invoked with open variables.

Quelle: Wiebke Petersen, Grunkurs Prolog, HHU Düsseldorf, https://user.phil.hhu.de/~petersen/wise2324_Prolog/WiSe2324_Prolog

Multiple Answers to a Query

- The inference procedure of Prolog computes one solution.
- The user can force the system to compute the next solution by typing a „;“ (typing „;“ is interpreted by the system as a fail and thus backtracking is started to compute an alternative solution)
- Example:

```
father(peter,mary) .  
father(peter,john) .  
father(peter,paul) .
```

```
sibling(Y,Z) :- father(X,Y) , father(X,Z) .  
sibling(Y,Z) :- mother(X,Y) , mother(X,Z) .
```

```
?- sibling(X,Y) .  
X=mary, Y=mary;  
X=mary, Y=john;  
X=mary, Y=paul;  
X=john, Y=mary
```


Negation and Cut

Negation as Failure

- Prolog allows a form of negation that is called negation as failure

- A negated query

not Q

is considered proved if the system fails to prove Q

- Thus, the clause

alive(X) :- not dead(X)

can be read as „Everyone is alive if not provably dead“

The Cut Operator

- Under procedural reading, a logic program consists of a set of procedures
- Each procedure consists of a sequence of alternatives
- The inference procedure of Prolog computes all possible alternatives for a query
- The cut operator (written as „!“) prevents backtracking. It is a special literal that is always true but that stops all other alternatives from being applied.

```
sibling(Y,Z) :- father(X,Y) , ! , father(X,Z) .  
sibling(Y,Z) :- mother(X,Y) , mother(X,Z) .
```

Application of the Cut

- The query `?-risk(X)` gives `X=high`

```
return(4) .  
  
risk(high) :- return(X), X < 5.  
risk(low) .
```

- When asking for another solution is also give `X=low`, which is wrong.
- How this be avoided?

Example with Cut

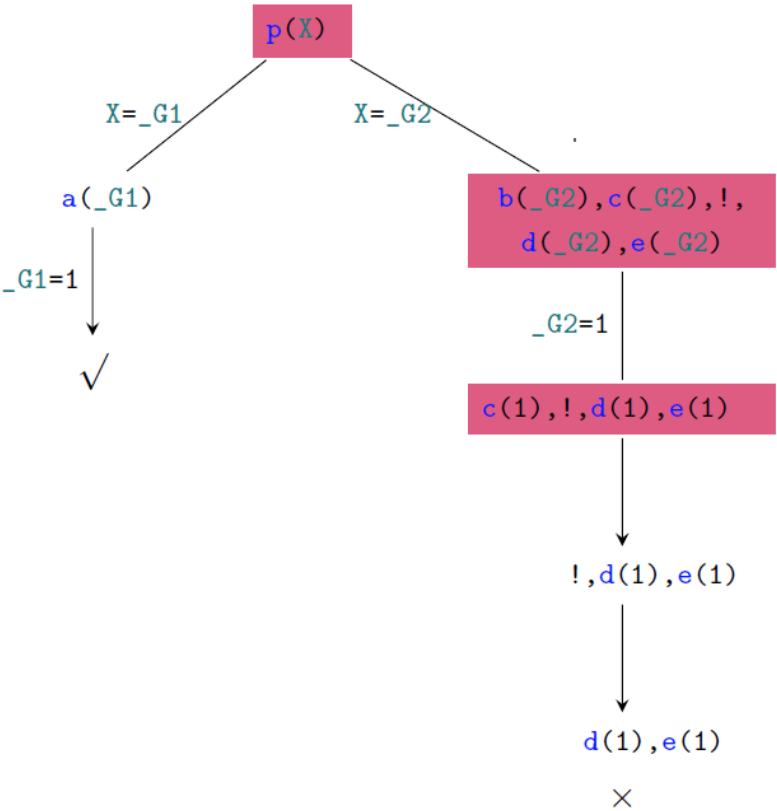
```

p(X) :- a(X) .
p(X) :- b(X) ,
        c(X) , ! ,
        d(X) ,
        e(X) .
p(X) :- f(X) .

a(1) .
b(1) .
b(2) .
c(1) .
c(2) .
d(2) .
e(2) .
f(3) .
    
```

```

?-p(X) .
X=1;
false.
    
```



Defining Negation as Failure with the Cut Operator

- The cut operator can be used to define negation as failure

```
not(Q) :- Q, !, fail.  
not(Q) .
```

- If ?- Q can be proved then the query not(Q) fails.
- If Q cannot be proved, the second clause is applied which always succeeds.
- If Q can be proved the second clause must not be applied. This is assured by the cut: If Q can be proved, then the cut prevents backtracking.

Using the Cut

■ Reasons to use the Cut

- ◆ Efficiency: Cropping the search space
- ◆ Shorter Programs
- ◆ Enforcing determinism.
- ◆ Modeling of defaults.

Use the cut only if there is an obvious advantage.

■ Downsides of the cut

- ◆ The cut destroys the declarativity of prolog programs.
- ◆ The interpretation of a predicate definition with cuts is usually only possible if the order of the proof steps is taken into account.



Arithmetics

Built-in Arithmetic: The Operator is/2

- In Prolog there is a set of built-in functions for arithmetics. To apply these function there exists a special predicate „is“:

X is Y is true when X is equal to the value of Y.

- Built-in functions include: +, −, *, /, //, mod, (// performs integer division)
 - ◆ Using these functions we can compute a value for terms involving numbers.
- Example:
 - ◆ `?- X is 7+1.`

Will give the answer **X = 8**

- The **is** Predicate works as follows:
 - ◆ First evaluate the right-hand argument (after the „is“)
 - ◆ The result is unified with the left-hand argument.
 - ◆ The values of all the variables on the right-hand side of is must be known for evaluation to succeed.

How to use is/2

- The operator forces the second argument to be evaluated immediately. Therefore, the second argument must be an evaluable arithmetic expression
- If the second argument cannot be evaluated, Prolog aborts with an error message

```
?- 3+5 is X.
```

```
?- X is 4+Y.
```

```
?- X is a.
```

Comparison

The comparison operators $<$ (smaller), $=<$, (less than or equal to), $>$ (greater), $>=$ (greater than or equal to), $==$ (equal), and \neq (unequal) force the immediate evaluation of both arguments

```
?- 1+4 < 3*5.
```

```
?- 1+7 =< 3*2.
```

```
?- 1+3 == 2*2.
```

```
?- 1+3 \= 2*3.
```

```
?- X < 3.
```

Comparison

Equality:

Pred	Description	Variable Substitution	Arithmetic Computation
=	unifiable	yes	no
is	is value of	first	second
==	same value	no	yes
===	identical	no	no

Other Comparisons:

$X > Y$	The value of X is greater than the value of Y
$X \geq Y$	The value of X is greater than or equal to the value of Y
$X < Y$	The value of X is less than the value of Y
$X \leq Y$	The value of X is less than or equal to the value of Y
$X \neq Y$	The values of X and Y are unequal



Lists

Lists in Prolog

- Lists are very powerful data structures in Prolog.
- Lists are finite sequences of elements
- Lists can contain different types of terms
- Lists can be nested (lists can have lists as items)
- Difference to sets:
 - ◆ The order of the elements is important $[a,b,c] \neq [b,a,c]$
 - ◆ The same item can appear multiple times in a list

```
[mia, vincent, jules, mia]
[mia, 2, mother(jules), x, 1.7]
[]
[mia, [[3,4,paul], mother(jules)], x]
```

Unification of Lists

- Two lists are unifiable,
 - ♦ if they are of the same length, and
 - ♦ if the corresponding list items are unifiable.
- The length of a list is the number of items it contains

```
?- [a,b,X]=[Y,b,3].  
X = 3, Y = a
```

```
?- [[a,b,c],b,3]=[Y,b,3].  
Y = [a, b, c]
```

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

```
?- [a,b,X,c]=[Y,b,3].  
false.
```

```
?- [a,c,3]=[Y,b,3].  
false.
```

Listenzerlegung in Prolog

- The list constructor `'|'` divides a list in head and tail.
 - ◆ The head is the first element of the list
 - ◆ The tail is the rest of the list. It is itself a list

```
?- [Head|Tail] = [mia, vincent, jules, mia].  
Head = mia,  
Tail = [vincent, jules, mia].
```

- A empty list has no head and therefore cannot be split

```
?- [Head|Tail] = [].  
false.
```

- The `'|'` can also separate more than one leading elements

```
?- [First,Second|Tail] = [mia, vincent, jules, mia].  
First = mia,  
Second = vincent,  
Tail = [jules, mia].
```


Anonymous Variable

- The variable ‘_’ is the anonymous variable in Prolog.
- It is always used when a value is no longer needed later.
- Unlike other variables, each occurrence of the anonymous variable is independent of the others. So it can be initialized differently again and again:

```
?- [_,x2,_,x4|_] = [mia, vincent, jules, mia, otto, lena].  
x2 = vincent,  
x4 = mia.
```

Task

- Define a predicate `member/2`, which tests whether an element occurs in a list

Note: `member/2` is a predefined predicate in some prolog systems that is loaded automatically. To define our own predicate, we use our own name, e.g. `my_member/2`.

Predicate member/2

- member/2 is a recursively defined predicate that checks whether an item appears in a list:

```
% member/2, member(Term, List)
member(X, [X|_]) .
member(X, [_|T]) :- member(X, T) .
```

- The fact member(X,[X|_]). says that something is an element of a list if it is the first item (the head) of the list.
- The rule member(X,[_|T]):- member(X,T). says that something is an element of a list if it is an element of the remainder list (the tail).
- Each item in a list is either the first item or an item in the tail

Declarative Application of member/2

- member/2 can be used to
- test whether an element occurs in a list (the first argument is a constant)

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

- find the elements of a list (the first argument is a variable)

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

Basic Predicates for List Manipulation

Four predicates for recursive lists processing:

member/2 access to list elements.

`member (Element, List)`

append/3 concatenation of lists.

`append (List1, List2, Konkatlist)`

delete/3 deleting list elements or adding elements to a list.

`delete (Element, List, ListDeleted)`

reverse/2 reverting a list

`reverse (List, ListReversed)`

Concatenating Lists: append/3

```
append([], L, L) .  
append([H|T1], L2, [H|T3]) :- append(T1, L2, T3) .
```

The predicate **append/3** can

- **test**, whether a list is the concatenation of two lists:

```
?- append([1,2,3], [4,5,6], [1,2,3,4,5,6]) .  
true.
```

- **concatenate** two lists :

```
?- append([1,2,3], [4,5,6], L) .  
L = [1,2,3,4,5,6].
```

- **divide** lists :

```
■ ?- append(L, [4,5,6], [1,2,3,4,5,6]) .  
L = [1,2,3] .  
  
?- append([1,2,3], L, [1,2,3,4,5,6]) .  
L = [4,5,6] .
```

```
?- append(X, Y, [a,b,c]) .  
X = [], Y = [a,b,c] ;  
X = [a], Y = [b,c] ;  
X = [a,b], Y = [c] ;  
X = [a,b,c], Y = [] ;
```

Suffix, Prefix and Sublist

The predicate `append/3` can be used to determine sublists:

- **Prefix** of a list `[a,b,c,d]`: `[],[a],[a,b],[a,b,c],[a,b,c,d]`

```
prefix(P,L) :- append(P,_,L) .
```

- **Suffix** of a list `[a,b,c,d]`: `[],[d],[c,d],[b,c,d],[a,b,c,d]`

```
suffix(S,L) :- append(_,S,L) .
```

- **Sublists** of a list `[a,b,c]`: `[],[a],[a,b],[a,b,c],[b],[b,c],[c]`

```
sublist(SL, L) :- prefix(P,L) , suffix(SL,P) .
```

Delete a list element: delete/3

```
% delete/3, delete(Term,List1,List2)

delete(X,[X|T],T) .
delete(X,[H|T1],[H|T2]) :-
    delete(X,T1,T2) .
```

- **delete/3** relates a term and two lists such that list2 is the result of deleting one occurrence of the term in list 1

```
?- delete(b,[a,b,c],[a,c]) .
true.

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

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

```
?- delete(1,X,[a,b,c]) .
X = [1, a, b, c] ;
X = [a, 1, b, c] ;
X = [a, b, 1, c] ;
X = [a, b, c, 1] .
```


Length of a List

```
% len1/2 , len1(List, Length)

length([],0) .
length([_|T],N):- length(T,X) ,
                  N is X+1.
```

```
?- len1([a,[b,e,[f,g]],food(cheese),X],4) .

true.

?- len1([a,b,a],X) .

X=3.
```