$\boxed{\text{Predicate Logic}}$

$\boxed{\text{Why is logic important in AI?}}$

Predicate Logic

- Propositional logic is included in predicate logic.
- **Predicate Logic** is also called **First order predicate calculus**, *First order Logic*, and *Predicate Calculus*.
- Predicate calculus sentences are built on predicates, variables, terms, functions and connectives. They are evaluated to true or false.
  - A **predicate** "is a quantified proposition with variables".
  - Quantifiers are $\forall$ (for all) and $\exists$ (exists).
  - T and F are predicates.
  - Variables are x, y, z etc.
  - Functions are of the form $f(x1, x2, .., xn)$. f is of arity n
  - Constants are functions of arity 0.
  - Terms are either constants, variables or function expressions.
  - Connectives are $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\iff$. They are used to create predicate sentences.

$\boxed{\text{Examples}}$

- $p(x)$
- p (refer to propositional logic)
- π (a constant)
- $\geq (x, y)$ (for $x \geq y$)
- $= (x, y)$ (for $x = y$)
- $= (f(x), z)$ (for $f(x) = z$)
- $parentof(x, y)$ is the same as $\forall x, \forall y, parentof(x, y)$
- $fatherof(x, y)$
- $speaks(x, y)$
- $prime(n)$
- $\forall x(speaks(x, Japanese))$
- $\exists x(speaks(x, Japanese))$
- $\forall x \exists y(speaks(x, y))$

- $\exists ! x(cannotread(x)$
- $father(x, y) \wedge man(x)$
- $p(x, y) \rightarrow (\exists z)p(x, z) \wedge p(z, y)$

$\boxed{\text{Aristotle Syllogism}}$

Problem

All men are mortal.

Socrates is a man.

Therefore, Socrates is mortal.

$$\downarrow$$

Model the problem

Hypotheses:

$$man(x) \rightarrow mortal(x)$$

$$man(socrates)$$

Goal to prove:

$$mortal(Socrates)$$

$$\downarrow$$

Proof

Deduction / Theorem Proving

Semantics

- One of the important tasks in predicate logic is to provide meaning to the expressions.

- A predicate is **satisfiable** if for some particular assignment of values to its variables (**interpretation**) the predicate is true. A **domain** needs to be considered for the values.

- A predicate is **valid** if for all assignment of values to its variables the predicate is true.

- *Interpretation* and *validity* can be generalized to predicate logic sentences.

- Examples:
  - $p(x,y) \to (\exists z)p(x,z) \wedge p(z,y)$
    * Interpretation 1: We interpret p as $<$ and the domain as the real numbers. We can pick $Z = (X+Y)/2$
    * Interpretation 2: We interpret p as true for the pairs $aa, ab, ba, bc, cb,$and c on the domain $\{a,b,c\}$.
  - $p(x) \vee \neg p(x)$ is valid

$$\boxed{\textbf{Resolution}}$$

- The resolution inference rule in propositional logic is the following:

  From $\leftarrow p_1, \ldots, p_n$ *and* $p_1 \leftarrow q_1, \ldots, q_k$
  derive $\leftarrow q_1, \ldots, q_k, p_2, \ldots, p_n$.

$$\begin{array}{l} \leftarrow p_1, \ldots, p_n \\ p_1 \leftarrow q_1, \ldots, q_k \\ \hline \therefore \leftarrow q_1, \ldots, q_k, p_2, \ldots, p_n \end{array}$$

- In the case with variables, we need to unify predicates and terms.

Results

- There is a distinction between what is provable and what is true by proof systems.

  Sound / correct: If something is provable, then it is true. (required)

  Complete: If something is true, then it is provable. (optional)

- Goedel's theorem states that if we take expressions describing number theory (i.e., arithmetic for the nonnegative integers) as hypotheses, then for any proof system, there is some expression that is entailed by the hypotheses but cannot be proved from them.

- Predicate logic is complete.

- Turing's theorem describes a formal model of a computer called a "Turing machine" and says that there are problems that cannot be solved by a computer. Predicate logic is undecidable. Some domain-specific problems can be solved.

PROLOG

Paradigm

- Declarative programming paradigm
  - The programmer declares the goals of the computation rather than the detailed algorithm by which these goals can be achieved.
- Logic programming is based on:
  - unification (Robinson, 1965) and
  - resolution (Robinson, 1965)
- Two important features of logic programming are:
  - non-determinism and
  - backtracking
- Popular in artificial intelligence
- Applications:
  - Natural language processing
  - Theorem proving
  - Databases
  - Expert systems
- PROLOG is a logic programming language (Colmerauer, 1972)

Logics (Review)

- **Propositional Logic**
  - Propositions:
    - true and false are propositions.
    - Propositional variables are propositions.
    - If p and q are propositions, then:
      - $p \wedge q$, $p \vee q$, $p \rightarrow q$, $p \leftrightarrow q$ and $\neg p$

      are propositions.
    - Precedence on connectives: $\neg > \wedge > \vee > \rightarrow > \leftrightarrow$
    - Examples: How do you formalize the following English sentences?
      - Provided that Marvin stays Nancy leaves.
      - Marvin stays but Nancy leaves.
      - Marvin stays although Nancy leaves.
  - Each proposition can be interpreted as either *true* or *false*.
    - Semantics methods
    - Syntactic methods
  - Propositional logic is decidable.

- **Predicate Logic - First order predicate calculus**
  - A **predicate** “is a quantified proposition with variables”.
  - Quantifiers are $\forall$ (for all) and $\exists$ (exists).
  - A predicate is **satisfiable** if for some particular assignment of values to its variables the predicate is true.
  - A predicate is **valid** if for all assignment of values to its variables the predicate is true.
  - Examples:
    * $parentof(x, y)$ is the same as $\forall x, \forall y, parentof(x, y)$
    * $fatherof(x, y)$
    * $speaks(x, y)$
    * $prime(n)$
    * $\forall x(speaks(x, Japanese))$
    * $\exists x(speaks(x, Japanese))$
    * $\forall x \exists y(speaks(x, y))$
  - The Incompleteness Theorem of Goedel, proven in 1930, demonstrates that first-order logic is in general undecidable.

$\boxed{\text{Normal Forms}}$

- *Normal forms* are equivalent formulas of a certain syntactic form. We consider **Conjunctive** and **disjunctive forms**.
- They permit us to answer certain questions more easily.
- A propositional formula is said to be in **conjunctive normal form** (CNF) if
  1. it contains only the logical connectives $\neg$, $\wedge$ and $\vee$,
  2. no logical connective occurs inside of a negation.
  3. no conjunction occurs inside of a disjunction.
- $(\neg p \vee q) \wedge (\neg p \vee \neg r \vee q)$ is a conjunctive normal form
- We speak of a **disjunctive normal form** (DNF) if the last condition is replaced by the condition that *no disjunction occur inside any conjunction*.
- $(\neg p \wedge q) \vee (p \wedge r)$ is a disjunctive normal form
- Any formula can be transformed to a CNF (or DNF).
- Exercise: Transform $\neg((p \vee q) \iff (P \rightarrow (q \wedge True)))$ into a CNF.

Clauses

- A **literal** is either a predicate or the negation of a predicate.
- Disjunctions of literals, $L_1 \vee \cdots \vee L_n$, are also called **clauses**.
- If a clause contains *at most* one positive literal, then it is called a **Horn clause**.
  - For example, $\neg p \vee \neg q$ and $\neg p \vee \neg q \vee r$ are Horn clauses, but $p \vee q$ is not a Horn clause.
- Horn clauses can be interpreted as program rules and used for computation, as it is done in **logic programming**.

Logic Program

- A **Horn clause** $\neg p_1 \vee \cdots \vee \neg p_n \vee q$ is logically equivalent to the implication $(p_1 \wedge \cdots \wedge p_n) \rightarrow q$.

- If the implication is known to be true, and one wishes to prove q, then it sufficient to show that $p_1, \ldots, p_n$ are all true; an observation that provides the logical basis for logic programming.

- A **logic program** is a set of Horn clauses, each containing exactly one positive literal (and zero or more negative literals). Such Horn clauses are usually written as backward implications

  $$q \leftarrow p_1, \ldots, p_n$$

  and called **program rules**. More specifically, q is called the **head** of the rule, and the sequence $p_1, \ldots, p_n$ the **body** of the rule.

- Each rule must have a head, but the body may be empty and in that case the rule is called a **fact**. For instance $q \leftarrow$ is a fact.

- A logic program is composed of rules and facts.

$$\boxed{\textbf{Notations}}$$

- A Horn clause is a rule and it is written as:

  $$q \leftarrow p_1, \ldots, p_n$$

  It means the same as:

  $$\neg p_1 \vee \cdots \vee \neg p_n \vee q$$

- If $n = 0$, the clause is a fact and is written: $q \leftarrow$.
  $q \leftarrow$ is the same as q.

- $\leftarrow p$ is the negation of the goal (the query) and it is the same as $\neg p$.

Logic program

Propositional case

$e \leftarrow$
$f \leftarrow$
$b \leftarrow$
$c \leftarrow a, b$
$a \leftarrow e, f$

- is a propositional logic program of 5 rules. The first 3 rules have an empty body and represent **facts**.
- In addition to the program rules one needs to specify a **goal** (or a list of goals) that we want to prove.

  Example: If we want to prove c, the goal is c.
- A computation with a logic program represents an attempt to derive the goal from the program rules (in an indirect way by deriving a **contradiction** in the form of the "empty clause " (represented by $\square$) from the **negation** of the goal).
- The logical inference rule underlying such computations is called **resolution**.

$\boxed{\textbf{Logic program}}$

With variables

p(edward7, george5) $\leftarrow$
p(victoria, edward7)$\leftarrow$
p(alexandra, george5)$\leftarrow$
p(george6, elizabeth2)$\leftarrow$
p(george5, george6)$\leftarrow$
$g(X, Y) \leftarrow p(X, Z), p(Z, Y)$

- is a logic program of 6 rules. The first 5 rules have an empty body and represent facts (about the British royal family).

- The last rule defines the *grandparent relation* in terms of the *parent relation*: a person X is a grandparent of Y if there is a third person Z, such that X is the parent of Z, and Z the parent of Y.

- Informally, the rule $g(X, Y) \leftarrow p(X, Z), p(Z, Y)$ may be thought of as a schema representing all clauses obtained by substituting specific values for the variables, e.g.,

  $g(victoria, george5) \leftarrow p(victoria, edward7), p(edward7, george5$

  X = victoria, Z = edward7, Y = george5

- In addition to the program rules one needs to specify a **goal** (or a list of goals) that we want to prove.

  Example: If we want to prove that the grandfather of George V is Victoria then the goal is g(victoria, george5).

- A computation with a logic program represents an attempt to derive the goal from the program rules (in an indirect way by deriving a **contradiction** in the form of the "empty clause" ($\Box$) from the **negation** of the goal).

- The logical inference rule underlying such computations is called **resolution**.

$\boxed{\textbf{Unification}}$

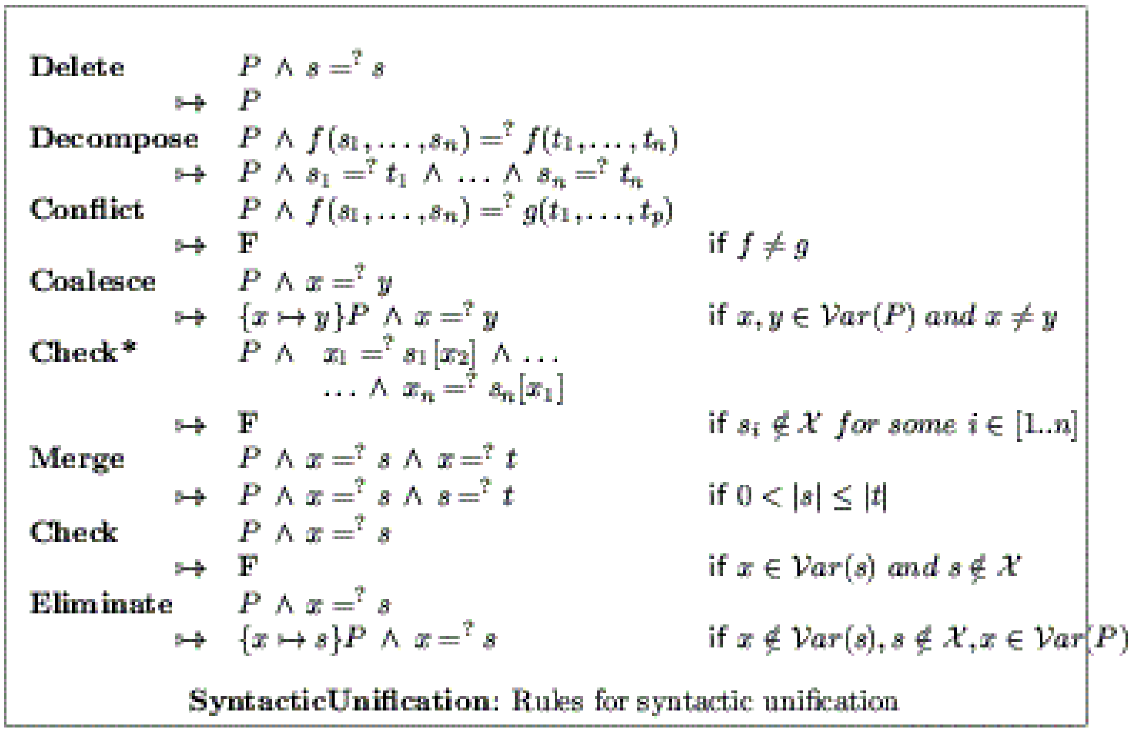
- **Unification** is a pattern-matching process that determines what particular instantiation can be made to variables to make two predicates equal. This instantiation is called a **substitution**.

- Examples:

  - How to make $brotherof(john, X)$ and $brotherof(Y, bill)$ equal?
    With the substitution: $X \mapsto bill$, $Y \mapsto john$

  - How to make b and b equal?
    With the substitution: id (identity)

Unification algorithm



Delete	$P \wedge s =^? s$	
	$\mapsto P$	
Decompose	$P \wedge f(s_1, \dots, s_n) =^? f(t_1, \dots, t_n)$	
	$\mapsto P \wedge s_1 =^? t_1 \wedge \dots \wedge s_n =^? t_n$	
Conflict	$P \wedge f(s_1, \dots, s_n) =^? g(t_1, \dots, t_p)$	
	$\mapsto \mathbf{F}$	if $f \neq g$
Coalesce	$P \wedge x =^? y$	
	$\mapsto \{x \mapsto y\}P \wedge x =^? y$	if $x, y \in \mathcal{V}ar(P)$ *and* $x \neq y$
Check*	$P \wedge x_1 =^? s_1[x_2] \wedge \dots$	
	$\dots \wedge x_n =^? s_n[x_1]$	
	$\mapsto \mathbf{F}$	if $s_i \notin \mathcal{X}$ *for some* $i \in [1..n]$
Merge	$P \wedge x =^? s \wedge x =^? t$	
	$\mapsto P \wedge x =^? s \wedge s =^? t$	if $0 < \lvert s \rvert \leq \lvert t \rvert$
Check	$P \wedge x =^? s$	
	$\mapsto \mathbf{F}$	if $x \in \mathcal{V}ar(s)$ *and* $s \notin \mathcal{X}$
Eliminate	$P \wedge x =^? s$	
	$\mapsto \{x \mapsto s\}P \wedge x =^? s$	if $x \notin \mathcal{V}ar(s), s \notin \mathcal{X}, x \in \mathcal{V}ar(P)$

SyntacticUnification: Rules for syntactic unification

$\boxed{\textbf{Resolution}}$

Propositional case

- The propositional version of resolution for Horn clauses is:

  From $\leftarrow p_1, \ldots, p_n$ *and* $p_1 \leftarrow q_1, \ldots, q_k$
  derive $\leftarrow q_1, \ldots, q_k, p_2, \ldots, p_n$.

$$\begin{array}{l} \leftarrow p_1, \ldots, p_n \\ p_1 \leftarrow q_1, \ldots, q_k \\ \hline \therefore \leftarrow q_1, \ldots, q_k, p_2, \ldots, p_n \end{array}$$

  – What is the rule if $n = 1$ and $k = 1$? It's the Modus Ponens.

$$\begin{array}{l} \leftarrow p_1 \\ p_1 \leftarrow q_1 \\ \hline \therefore \leftarrow q_1 \end{array}$$

  – What is the rule if $n = 1$ and $k = 0$?

$$\begin{array}{l} \leftarrow p_1 \\ p_1 \leftarrow \\ \hline \therefore \Box \end{array}$$

- **Example:** Assume we want to prove c.
  - The negation of the goal c is written as a negative clause
    $$\leftarrow c.$$
  - We have also seen that c is the head of a rule $(c \leftarrow a, b)$.
  - This indicates that the given goal may be *reduced* to subgoals (by the resolution rule)
    $$\leftarrow a, b.$$
  - We have also seen that a is the head of a rule $(a \leftarrow e, f)$.
  - This indicates that the given goal may be *reduced* to subgoals (by the resolution rule)
    $$\leftarrow e, f, b.$$
    where a is replaced by e, f.
- The three subgoals are present as facts and hence can be deleted, which results in the empty clause ($\square$).
- We conclude that the original goal logically follows from the program clauses.
- But much of the power of logic programming derives from the fact that resolution can be generalized to effectively handle clauses with variables.

$\boxed{\textbf{Resolution}}$

With variables

- Assume we want to prove that Victoria is the grand-mother of George.

- The negation of the above goal is written as a negative clause

$$\leftarrow g(victoria, george5).$$

- We have also seen that suitable values may be substituted for the variables in the last program rule, so that the head is g(victoria, george5) (X=victoria and Y = george5).

- This indicates that the given goal may be *reduced* to subgoals (backward reasoning)

$$\leftarrow p(victoria, edward7), p(edward7, george5).$$

- Both subgoals are present as facts and hence can be deleted, which results in the empty clause ($\Box$).

- We conclude that the original goal logically follows from the program clauses.

- Goals with variables are also possible.

  Example: If one specifies the goal

  $$\leftarrow g(victoria, X)$$

  the result of the computation will be a list of all grandchildren of Victoria. A discussion of these aspects of logical programming is beyond the scope of this course.

PROLOG

- SWI-prolog
  - Download Prolog here: https://www.swi-prolog.org
  - or use Prolog online here: https://swish.swi-prolog.org/example/examples.swinb
- Prolog files have $.pl$ as extensions
- Exemple: Let's consider the *likes.pl* file. There are 3 facts.

```
likes(john,mary).
likes(mary,sue).
likes(mary,tom).
```

- To run PROLOG type: *swipl*, then
- To load the *likes.pl* file, type: *[likes].* or *consult(likes)*..
- You can now play with prolog:

  Who are the people Mary likes?

```
likes(mary,X).
```

  X is a variable and must be written using a capital letter. Constants are written in lower cases.

To have all the solutions to the $likes(mary, X)$ goal, type n (for next) after each solution.

- You can also use swipl likes.pl to run likes directly.

- In PROLOG:
  - A variable begins with a capital letter.
  - A constant is written in lower cases.
  - Underscore characters are considered as variables.
  - All facts, rules and queries end with a period.

Prolog language

- Prolog reads the facts and rules in the order they are defined.
- Each clause is looked at from left to right.
- Numbers: 3, 2.5
- Strings: "" (e.g., "$Hello$")
- Assignment: is (e.g., X is 4+5.)
- Predefined functions: $-$, $+$, $*$, $/$, ^ , mod, abs, min, max, sign, random, sqrt, sin, cos, tan, log, exp (e.g., X is sin(pi/2).)
- Comparisons: =:=, \==, =\=, >, <, >=, =<
- Checking the types: var, nonvar, integer, float, number, atom, string (e.g., number(5))

Examples of programs

- Explicit definition 1:

```
f(x) = if x=0 then 1 else 5

PROLOG:
f(0,1).
f(X,5) :- X>0.
```

- Explicit definition 2:

```
f(x) = 2*x

PROLOG:
g(X,Y) :- Y is 2*X.
```

- Example:

  ```
  PROLOG:
  speaks(allen,russian).
  speaks(bob,english).
  speaks(mary,russian).
  speaks(mary,wnglish).
  talkswith(Person1,Person2):-speaks(Person1,L),
  speaks(Person2,L), Person1 \= Person2.
  ```

  How to know who talks with who?

- Recursive definition 1:

  ```
  fact(n) = if n=0 then 1 else n*fact(n-1)

  PROLOG:
  factorial(0,1).
  factorial(N,Result) :- N>0, M is N-1,
  factorial(M,SubResult), Result is N*SubResult.
  ```

- Recursive definition 2:

  ```
  fib(n) = if n=0 then 1 else if n=1 then 1
  else fib(n-1)+fib(n-2)

  PROLOG:
  fib(0,1).
  fib(1,1).
  fib(N,R) :- N>1, N1 is N-1, N2 is N-2, fib(N1,R1),
  fib(N2,R2), R is R1+R2.
  ```

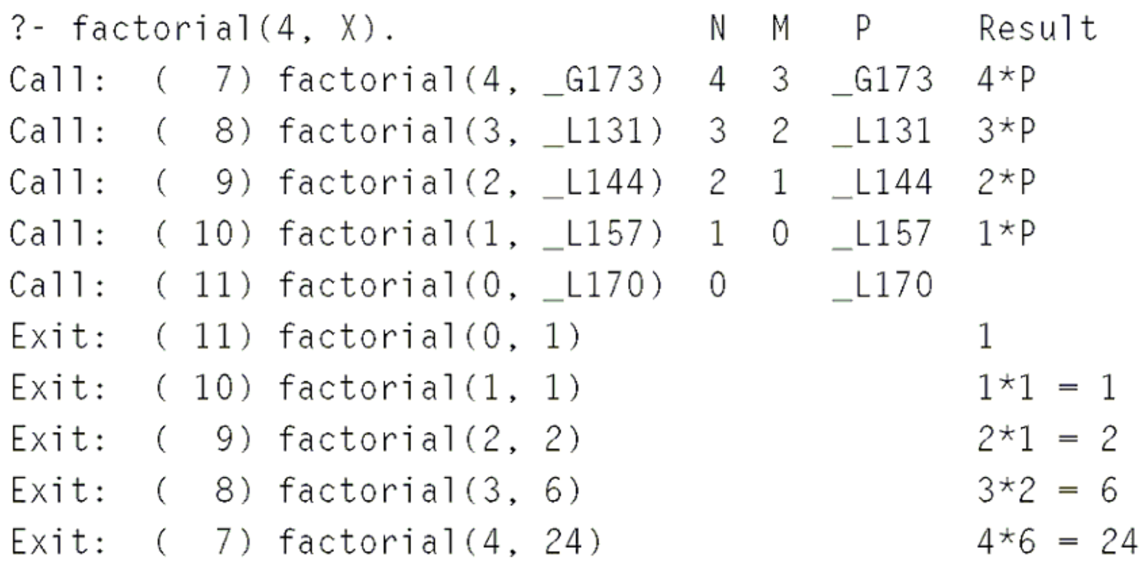
Tracing in PROLOG

- To trace a particular predicate p use:

  `trace(p/2). or trace, p/2`

- Example:

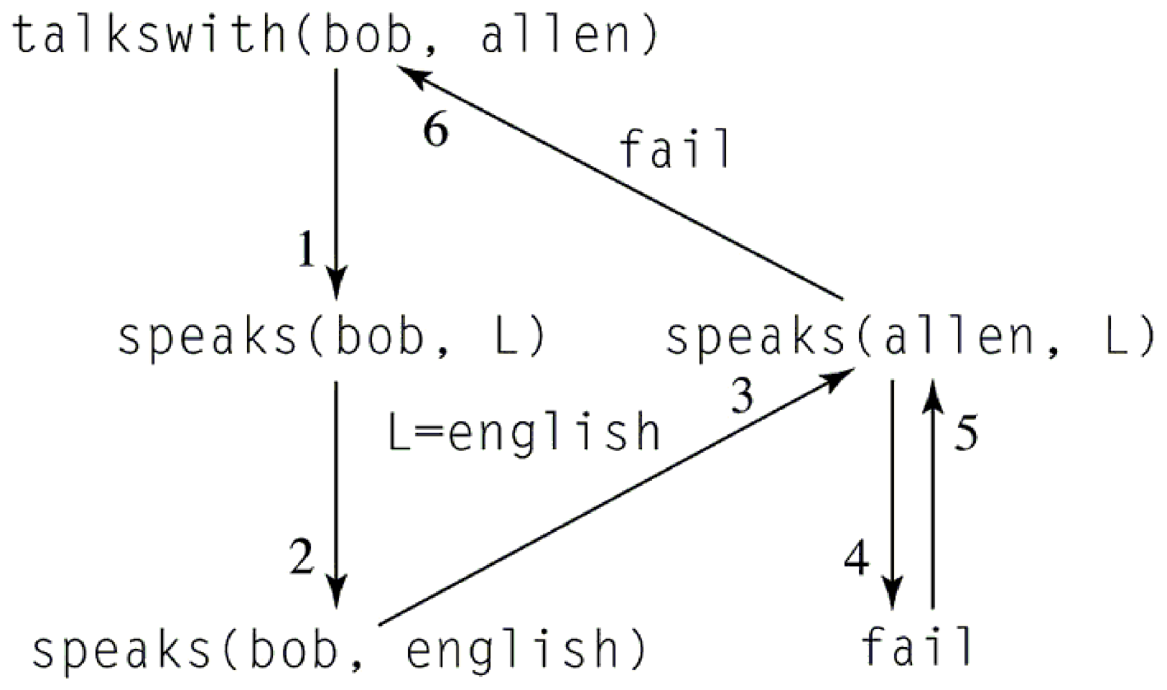
  `trace(factorial/2).`



```
?- factorial(4, X).                    N   M    P      Result
Call:   (  7) factorial(4, _G173)  4   3   _G173   4*P
Call:   (  8) factorial(3, _L131)  3   2   _L131   3*P
Call:   (  9) factorial(2, _L144)  2   1   _L144   2*P
Call:   ( 10) factorial(1, _L157)  1   0   _L157   1*P
Call:   ( 11) factorial(0, _L170)  0       _L170
Exit:   ( 11) factorial(0, 1)                      1
Exit:   ( 10) factorial(1, 1)                      1*1 = 1
Exit:   (  9) factorial(2, 2)                      2*1 = 2
Exit:   (  8) factorial(3, 6)                      3*2 = 6
Exit:   (  7) factorial(4, 24)                     4*6 = 24
```

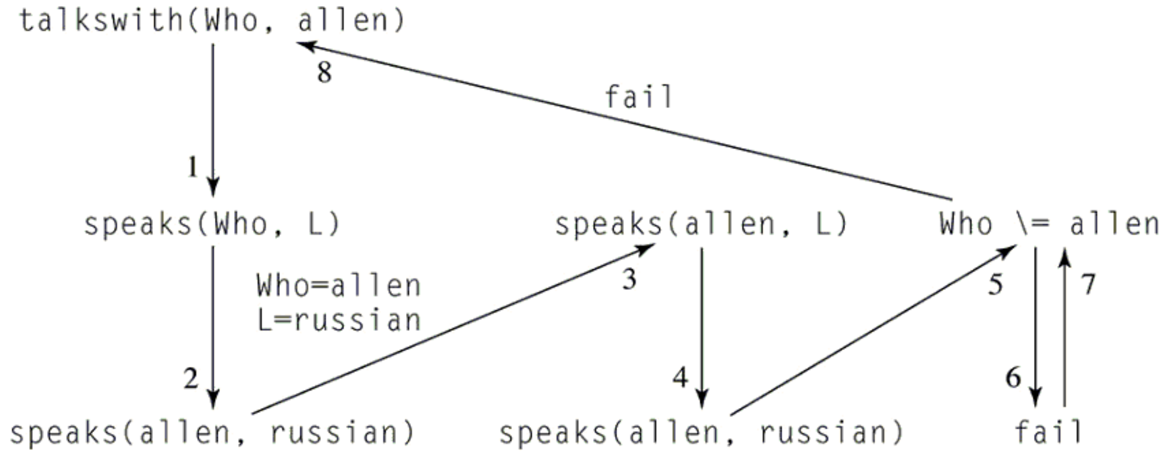
Goal without variables

```
talkswith(bob,allen).
```



Goal with variables

```
talkswith(Who,allen).
```



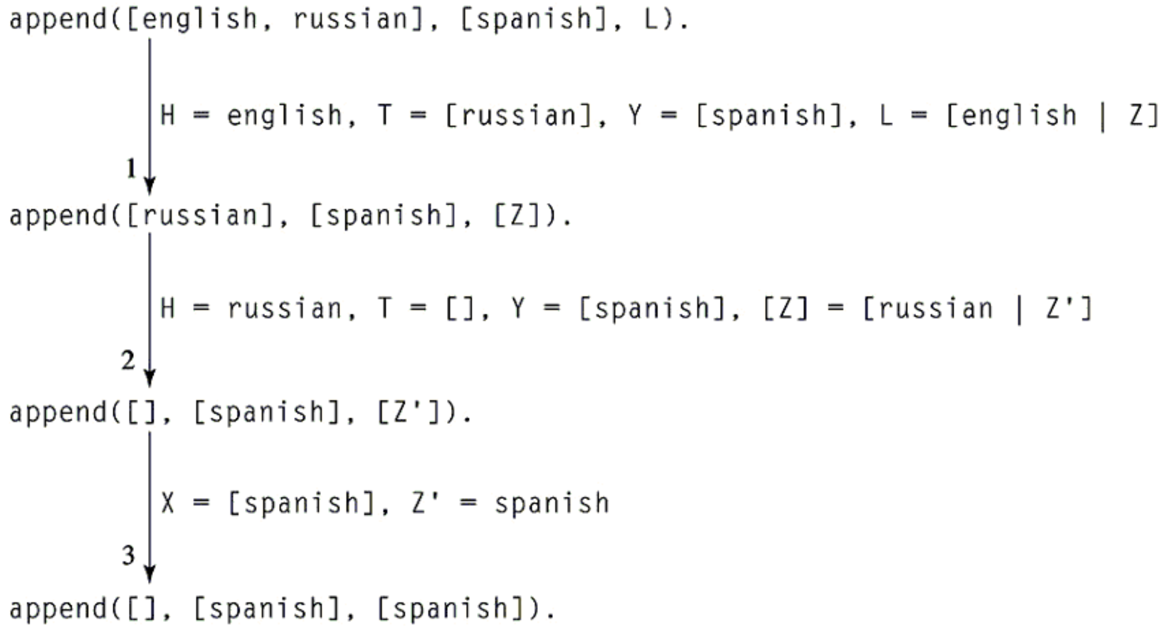
Lists in PROLOG

- The basic data structure in PROLOG is the *list*.
  - [] is the empty list
  - $[X, Y]$ is a list with 2 elements
  - $[\_, \_, Y]$ is a list with 3 elements
  - $[X|Y]$ denotes a list with head X and tail Y.
- Some built-in functions on lists:
  - $append(?List1, ?List2, ?List3)$
  - $length(?List1, ?Int)$
  - $reverse(+List1, -List2)$
  - $member(?Elem, ?List)$
  - $sort(+List, -Sorted)$ (to sort a list – it removes the duplicates)
- Definition of functions on lists:
  - member:

```
member1(X,[X|_]).
member1(X,[_|Y]) :- member1(X,Y).
```

- append:

```
append1([],X,X).
append1([H|T],Y,[H|Z]) :- append1(T,Y,Z).
```



```
append([english, russian], [spanish], L).
    |
    |  H = english, T = [russian], Y = [spanish], L = [english | Z]
  1 ↓
append([russian], [spanish], [Z]).
    |
    |  H = russian, T = [], Y = [spanish], [Z] = [russian | Z']
  2 ↓
append([], [spanish], [Z']).
    |
    |  X = [spanish], Z' = spanish
  3 ↓
append([], [spanish], [spanish]).
```

Cut

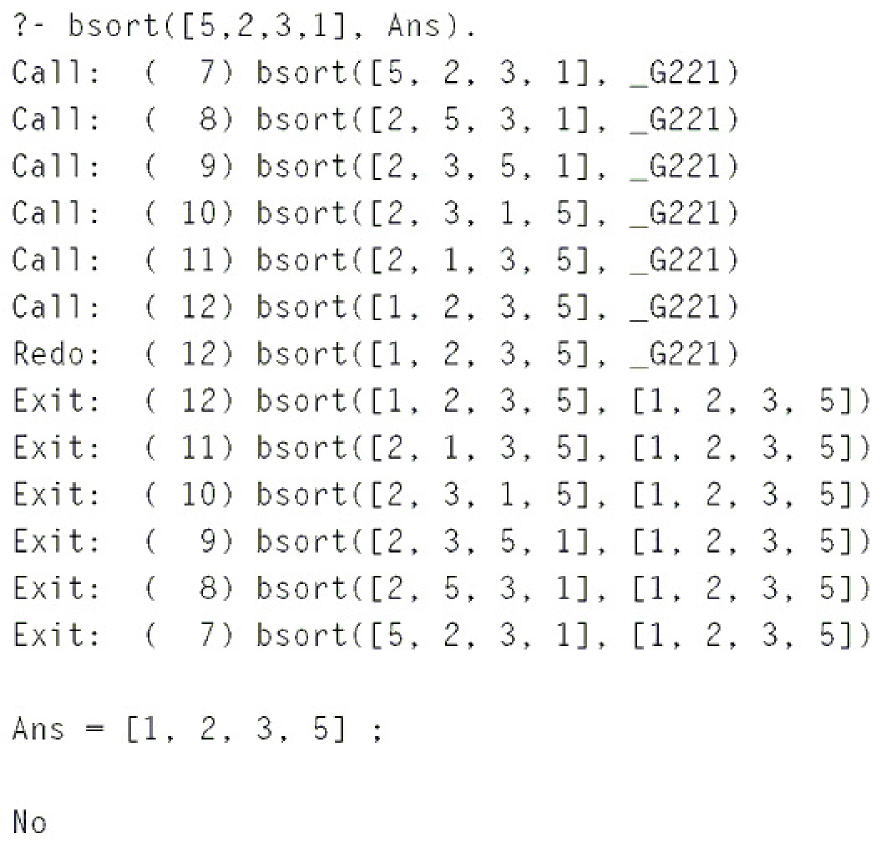
- The **cut** permits us to force the evaluation of a series of subgoals on the right-hand side of a rule not to be retried if the right-hand side succeeds once.
- You can thing about the cut as a *conditional statement*.
- The cut is implemented by !.
- Example 1:

```
f(x) = if x=0 then 1 else 5

PROLOG:
f(0,1).
f(X,5) :- X>0.
is the same as:
f(0,1) :- !.
f(X,5) :-.
```

- Example 2: Bubble Sort

```
bsort(L,S) :- append(U,[A,B|V],L), B<A, !,
append(U,[B,A|V],M), bsort(M,S).
bsort(L,L).
```



```
?- bsort([5,2,3,1], Ans).
Call:  (  7) bsort([5, 2, 3, 1], _G221)
Call:  (  8) bsort([2, 5, 3, 1], _G221)
Call:  (  9) bsort([2, 3, 5, 1], _G221)
Call:  ( 10) bsort([2, 3, 1, 5], _G221)
Call:  ( 11) bsort([2, 1, 3, 5], _G221)
Call:  ( 12) bsort([1, 2, 3, 5], _G221)
Redo:  ( 12) bsort([1, 2, 3, 5], _G221)
Exit:  ( 12) bsort([1, 2, 3, 5], [1, 2, 3, 5])
Exit:  ( 11) bsort([2, 1, 3, 5], [1, 2, 3, 5])
Exit:  ( 10) bsort([2, 3, 1, 5], [1, 2, 3, 5])
Exit:  (  9) bsort([2, 3, 5, 1], [1, 2, 3, 5])
Exit:  (  8) bsort([2, 5, 3, 1], [1, 2, 3, 5])
Exit:  (  7) bsort([5, 2, 3, 1], [1, 2, 3, 5])

Ans = [1, 2, 3, 5] ;

No
```

Natural language processing

- Write a program that is effectively a BNF grammar, which, when executed, will parse sentences in a natural language.

- Consider the following BNF grammar:

$s \to np\ vp$

$np \to det\ n$

$vp \to tv\ np$

$\quad \to iv$

$det \to \text{the}$

$\quad \to \text{a}$

$\quad \to \text{an}$

$n \to \text{giraffe}$

$\quad \to \text{apple}$

$iv \to \text{dreams}$

$tv \to \text{eats}$

$\quad \to \text{dreams}$

$$\begin{tikzcd}
 & & & s \arrow[dll] \arrow[dr] & \\
 & np \arrow[dl] \arrow[dr] & & & vp \arrow[d] \\
det \arrow[d] & & n \arrow[d] & & iv \arrow[d] \\
\text{the} & & \text{giraffe} & & \text{dreams}
\end{tikzcd}$$

Representation in Prolog 1

- We represent a sentence using a list.
- We can write Prolog rules that partition a sentence into its grammatical categories using the structure defined by the BNF grammar.
- For example:

  ```
  s -> np vp
  ```

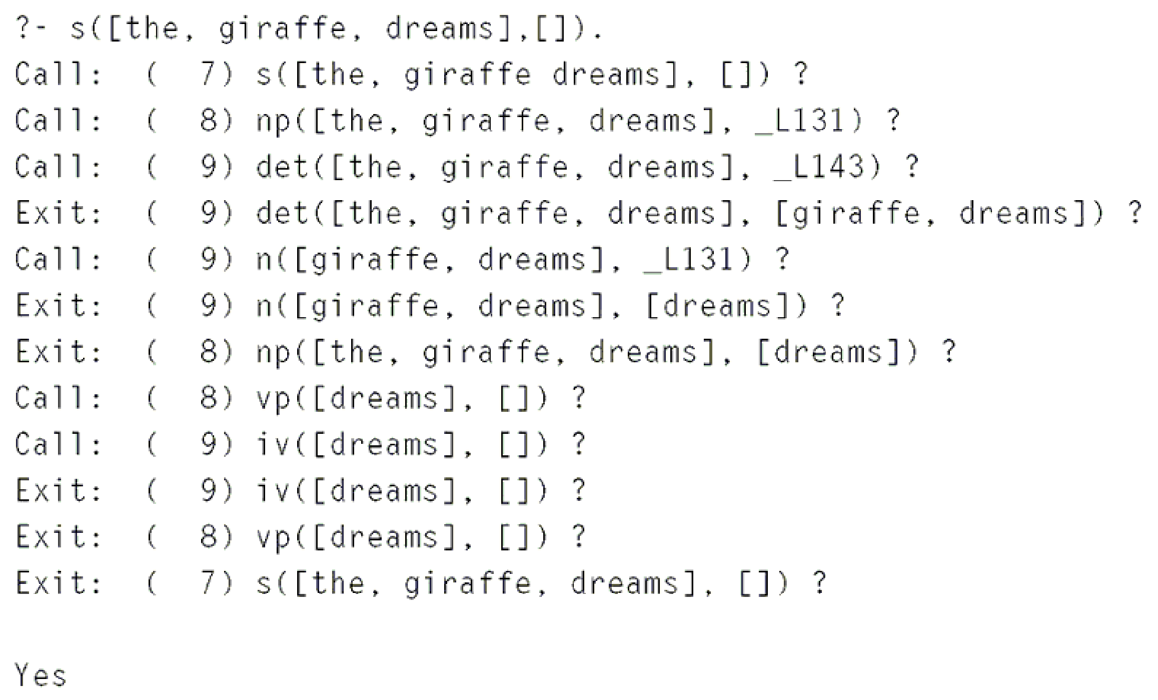
  is represented by:

  ```
  s(X,Y) :- np(X,U),vp(U,Y).
  ```

  X is the sentence being parsed and Y represents the resulting tail of the list that will remain to parse if this rule succeeds (to be applied).

- Assume we want to parse "the giraffe dreams". We write the query:

  `s([the,giraffe,dreams],X).`



```
?- s([the, giraffe, dreams],[]).
Call:  (  7) s([the, giraffe dreams], []) ?
Call:  (  8) np([the, giraffe, dreams], _L131) ?
Call:  (  9) det([the, giraffe, dreams], _L143) ?
Exit:  (  9) det([the, giraffe, dreams], [giraffe, dreams]) ?
Call:  (  9) n([giraffe, dreams], _L131) ?
Exit:  (  9) n([giraffe, dreams], [dreams]) ?
Exit:  (  8) np([the, giraffe, dreams], [dreams]) ?
Call:  (  8) vp([dreams], []) ?
Call:  (  9) iv([dreams], []) ?
Exit:  (  9) iv([dreams], []) ?
Exit:  (  8) vp([dreams], []) ?
Exit:  (  7) s([the, giraffe, dreams], []) ?

Yes
```

- Assume we want to parse "the giraffe sleeps".We write the query:

  `s([the,giraffe,sleeps],X).`

  The result will be "No".

- Assume we want all the sentences parsed by the grammar.

  `s(Sentence,Y).`

Representation in Prolog 2

- We use a notation called **Definite Clause Grammar** (DCG).
- This notation is close from the notation of context-free grammars rules.
- We use the operator $-->$ instead of $:-$.
- We remove the variables from the rules.
- But the meaning and the arity of the predicates do not change.
- DCG representation of the previous BNF grammar:

```
s --> np,vp.
np --> det,n.
vp --> iv.
vp --> tv,np.
det --> [the].
det --> [a].
n --> [giraffe].
n --> [apple].
iv --> [dreams].
tv --> [dreams].
tv --> [eats].
```

- Queries are the same as previously.

- If we modify slightly each rule we can add the capability to generate a parse tree directly from the grammar.

  This is done by adding an additional parameter to each rule and appropriate variables to hold the intermediate values that are derived.

  For example, the parse tree of “the giraffe dreams” can be represented by:

  ```
  s(np(det(the),n(giraffe)),vp(iv(dreams)))
  ```

  Here is the modified Prolog program:

  ```
  s(s(NP,VP)) --> np(NP),vp(VP).
  np(np(DET,N)) --> det(DET),n(N).
  vp(vp(VP)) --> iv(VP).
  vp(tv(TV),np(NP)) --> tv(TV),np(NP).
  det(det(the)) --> [the].
  det(det(a)) --> [a].
  n(n(giraffe)) --> [giraffe].
  n(n(apple)) --> [apple].
  iv(iv(dreams)) --> [dreams].
  tv(tv(dreams)) --> [dreams].
  tv(tv(dreams)) --> [eats].
  ```

  Here are some possible queries:

  ```
  s(Tree,[the,giraffe,dreams],X).
  s(Tree,Sentence,X).
  ```

DCG representation of the JAY language

```
expression --> conjunction, ['||'], expression ; conjunction.
conjunction --> relation, [&&], conjunction ; relation.
relation --> addition, [<], relation ;
             addition, [<=], relation ;
             addition, [>], relation ;
             addition, [>=], relation ;
             addition, [==], relation ;
             addition, ['!='],relation ;
             addition.
addition --> term, [+], addition ;
             term, [-], addition ;
             term.
term --> factor, [*], term ;
         factor, [/], term ;
         factor.
factor --> ['('], expression, [')'] ; [id] ; [lit].
```

Execution

```
expression(expression(C, '||', E)) --> conjunction(C), ['||'],
                                       expression(E).
expression(expression(C)) --> conjunction(C).
conjunction(conjunction(R, &&, C)) --> relation(R), [&&],
                                       conjunction(C).
conjunction(conjunction(R)) --> relation(R).
relation(relation(A, <, R)) --> addition(A), [<], relation(R).
relation(relation(A, <=, R)) --> addition(A), [<=], relation(R).
relation(relation(A, >, R)) --> addition(A), [>], relation(R).
relation(relation(A, >=, R)) --> addition(A), [>=], relation(R).
relation(relation(A, ==, R)) --> addition(A), [==], relation(R).
relation(relation(A,'!=',R)) --> addition(A),['!='],relation(R).
relation(relation(A)) --> addition(A).

addition(addition(T, +, R)) --> term(T), [+], addition(R).
addition(addition(T, -, R)) --> term(T), [-], addition(R).
addition(addition(T)) --> term(T).

term(term(F, *, T)) --> factor(F), [*], term(T).
term(term(F, /, T)) --> factor(F), [/], term(T).
term(term(F)) --> factor(F).

factor(factor('(', E, ')')) --> ['('], expression(E), [')'].
factor(factor(I)) --> identifier(I).
factor(factor(Val)) --> literal(Val).

identifier(id) --> [id].
literal(val) --> [val].
```