# Logic Programming

## An Introduction to Prolog

September 27, 2018

# Contents of this lecture

Part 1: An introduction in the logic programming language Prolog. Based largely on [Clocksin and Mellish, 2003].

# Contents of this lecture

Part 1: An introduction in the logic programming language Prolog. Based largely on [Clocksin and Mellish, 2003].

Part 2: A review of the theoretical basis of logic programming. Based on corresponding topics in [Ben-Ari, 2001] and [Nilsson and Maluszynski, 2000].

# Contents of this lecture

Part 1: An introduction in the logic programming language Prolog. Based largely on [Clocksin and Mellish, 2003].

Part 2: A review of the theoretical basis of logic programming. Based on corresponding topics in [Ben-Ari, 2001] and [Nilsson and Maluszynski, 2000].

Part 3: Advanced topics in logic programming/Prolog. Based on corresponding topics in [Ben-Ari, 2001] and [Nilsson and Maluszynski, 2000].

# Organizatorial items

- Lecturer: Isabela Drămnesc
- Course webpage:
  `http://staff.fmi.uvt.ro/~isabela.dramnesc`
  - 14 Courses
  - 7 Labs: working with Prolog
- Handouts: will be posted on the webpage of the lecture
- Grading:
  - 50% : 25% (weekly seminar assignments + activity + homework), 25% (the practical exam during the last lab)
  - 50% : 1 written exam (colloquy) at the end of the semester (during the last lecture)

# Recalling the Von Neumann machine

- The von Neumann machine (architecture) is characterized by:
  - large uniform store of memory,
  - processing unit with registers.
- A program for the von Neumann machine: a sequence of instructions for
  - moving data between memory and registers,
  - carrying out arithmetical-logical operations between registers,
  - control, etc.
- Most programming languages (like C, C++, Java, etc.) are influenced by and were designed for the von Neumann architecture.
  - In fact, such programming languages take into account the architecture of the machine they address and can be used to write efficient programs.
  - The above point is by no means trivial, and it leads to a separation of work ("the software crisis"):
  - finding solutions of problems (using reasoning),
  - implementation of the solutions (mundane and tedious).

# Alternatives to the von Neumann approach

- How about making programming part of problem solving?
- i.e. write programs as you solve problems?
- "rapid prototyping"?
- Logic programming is derived from an abstract model (not a reorganization/abstraction of a von Neumann machine).
- In logic programming
  - program = set of axioms,
  - computation = constructive proof of a goal statement.

# Logic programming: some history

- David Hilbert's program (early 20th century): formalize all mathematics using a finite, complete, consistent set of axioms.
- Kurt Gödel's incompleteness theorem (1931): any theory containing arithmetic cannot prove its own consistency.
- Alonzo Church and Alan Turing (independently, 1936): undecidability - no mechanical method to decide truth (in general).

# Logic programming: some history

- Alan Robinson (1965): the resolution method for first order logic (i.e. machine reasoning in first order logic).
- Robert Kowalski (1971): procedural interpretation of Horn clauses, i.e. computation in logic.
- Alan Colmerauer (1972): Prolog (PROgrammation en LOGique).prover.
- David H.D. Warren (mid-late 1970's): efficient implementation of Prolog.
- 1981 Japanese Fifth Generation Computer project: project to build the next generation computers with advanced AI capabilities (using a concurrent Prolog as the programming language).

# Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,
  - abstract problem solving,
  - natural language understanding,
  - symbolic equation solving,
  - design automation,
  - artificial intelligence,
  - biochemical structure analysis, etc.

# Applications of logic programming (cont'd)

- Industrial applications:
  - aviation:
    - SCORE - a longterm aiport capacity management system for coordinated airports (20% of air traffic worldwide, according to `www.pdc-aviation.com`)
    - FleetWatch - operational control, used by 21 international air companies.
  - personnel planning: StaffPlan (airports in Barcelona, Madrid; Hovedstaden region in Denmark).
  - information management for disasters: ARGOS - crisis management in CBRN (chemical, biological, radiological and nuclear) incidents – used by Australia, Brasil, Canada, Ireland, Denmark, Sweden, Norway, Poland, Estonia, Lithuania and Montenegro.

# Problem solving with Prolog

- Programming in Prolog:
  - rather than prescribe a sequence of steps to solve a problem,
  - describe known facts and relations, then ask questions.
- Use Prolog to solve problems that involve objects and relations between objects.
- Examples:
  - Objects: “John”, “book”, “jewel”, etc.
  - Relations: “John owns the book”, “The jewel is valuable”.
  - Rules: “Two people are sisters if they are both females and they have the same parents”.

# Problem solving with Prolog (cont'd)

- Attention!!! Problem solving requires modelling of the problem (with its respective limitations).
- Problem solving with Prolog:
  - declare facts about objects and their relations,
  - define rules about objects and their relations,
  - ask questions about objects and their relations.
- Programming in Prolog: a conversation with the Prolog interpreter.

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.
  - Prolog uses (mostly) prefix notation (but there are exceptions).

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.
  - Prolog uses (mostly) prefix notation (but there are exceptions).
  - Facts end with "." (full stop).

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.
  - Prolog uses (mostly) prefix notation (but there are exceptions).
  - Facts end with "." (full stop).
- A model is built in Prolog, and facts describe the model.

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.
  - Prolog uses (mostly) prefix notation (but there are exceptions).
  - Facts end with "." (full stop).

- A model is built in Prolog, and facts describe the model.
- The user has to be aware of the interpretation:

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

are not the same thing (unless explicitly specified).

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.
  - Prolog uses (mostly) prefix notation (but there are exceptions).
  - Facts end with "." (full stop).

- A model is built in Prolog, and facts describe the model.
- The user has to be aware of the interpretation:

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

  are not the same thing (unless explicitly specified).
- Arbitrary numbers of arguments are allowed.

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.
  - Prolog uses (mostly) prefix notation (but there are exceptions).
  - Facts end with "." (full stop).

- A model is built in Prolog, and facts describe the model.
- The user has to be aware of the interpretation:

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

  are not the same thing (unless explicitly specified).

- Arbitrary numbers of arguments are allowed.
- Notation: likes /2 indicates a binary predicate.

# Facts

- Stating a fact in Prolog:

```
likes(johnny, mary).
```

  - Names of relations (predicates) and objects are written in lower case letter.
  - Prolog uses (mostly) prefix notation (but there are exceptions).
  - Facts end with "." (full stop).

- A model is built in Prolog, and facts describe the model.
- The user has to be aware of the interpretation:

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

  are not the same thing (unless explicitly specified).

- Arbitrary numbers of arguments are allowed.
- Notation: likes /2 indicates a binary predicate.
- Facts are part of the Prolog database (knowledge base).

# Queries

- A query in Prolog:

```
?- owns(mary, book).
```

  Prolog searches in the knowledge base for facts that match the question:

# Queries

- A query in Prolog:

```
?- owns(mary, book).
```

  Prolog searches in the knowledge base for facts that match the question:
- Prolog answers true if :

# Queries

- A query in Prolog:

```
?- owns(mary, book).
```

  Prolog searches in the knowledge base for facts that match the question:
- Prolog answers true if :
  - the predicates are the same,

# Queries

- A query in Prolog:

```
?- owns(mary, book).
```

  Prolog searches in the knowledge base for facts that match the question:

- Prolog answers true if :
  - the predicates are the same,
  - arguments are the same,

# Queries

- A query in Prolog:

```
?- owns(mary, book).
```

  Prolog searches in the knowledge base for facts that match the question:
- Prolog answers true if :
  - the predicates are the same,
  - arguments are the same,
- Orthерwise the answer is false :

# Queries

- A query in Prolog:

```
?- owns(mary, book).
```

  Prolog searches in the knowledge base for facts that match the question:
- Prolog answers true if :
  - the predicates are the same,
  - arguments are the same,
- OrthERwise the answer is false :
  - only what is known is true (“closed world assumption”),

# Queries

- A query in Prolog:

  ```
  ?- owns(mary, book).
  ```

  Prolog searches in the knowledge base for facts that match the question:
- Prolog answers true if :
  - the predicates are the same,
  - arguments are the same,
- OrthERwise the answer is false :
  - only what is known is true ("closed world assumption"),
  - Attention: false may not mean that the answer is false (but more like "not derivable from the knowledge").

# Variables

- Think variables in predicate logic.

# Variables

- Think variables in predicate logic.
- Instead of:

```
?- likes(john, mary).
?- likes(john, apples).
?- likes(john, candy).
```

ask something like “What does John like?” (i.e. give everything that John likes).

# Variables

- Think variables in predicate logic.
- Instead of:

```
?- likes(john, mary).
?- likes(john, apples).
?- likes(john, candy).
```

  ask something like "What does John like?" (i.e. give everything that John likes).
- Variables stand for objects to be determined by Prolog.

# Variables

- Think variables in predicate logic.
- Instead of:

```
?- likes(john, mary).
?- likes(john, apples).
?- likes(john, candy).
```

  ask something like "What does John like?" (i.e. give everything that John likes).
- Variables stand for objects to be determined by Prolog.
- Variables can be:

# Variables

- Think variables in predicate logic.
- Instead of:

```
?- likes(john, mary).
?- likes(john, apples).
?- likes(john, candy).
```

  ask something like "What does John like?" (i.e. give everything that John likes).
- Variables stand for objects to be determined by Prolog.
- Variables can be:
  - instantiated - there is an object the variable stands for,

# Variables

- Think variables in predicate logic.
- Instead of:

```
?- likes(john, mary).
?- likes(john, apples).
?- likes(john, candy).
```

  ask something like "What does John like?" (i.e. give everything that John likes).

- Variables stand for objects to be determined by Prolog.
- Variables can be:
  - instantiated - there is an object the variable stands for,
  - uninstantiated - it is not (yet) known what the variable stands for.

# Variables

- Think variables in predicate logic.
- Instead of:

```
?- likes(john, mary).
?- likes(john, apples).
?- likes(john, candy).
```

  ask something like “What does John like?” (i.e. give everything that John likes).

- Variables stand for objects to be determined by Prolog.
- Variables can be:
  - instantiated - there is an object the variable stands for,
  - uninstantiated - it is not (yet) known what the variable stands for.
- In Prolog variables start with CAPITAL LETTERS:

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

# Prolog computation: example

- Consider the following facts in a Prolog knowledge base:

```
...
likes(john, flowers).
likes(john, mary).
likes(paul, mary).
...
```

# Prolog computation: example

- Consider the following facts in a Prolog knowledge base:

```
...
likes(john, flowers).
likes(john, mary).
likes(paul, mary).
...
```

- To the query

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

Prolog will answer

```
X = flowers
```

and wait for further instructions.

# Prolog answer computation

- Prolog searches the knowledge base for a fact that matches the query,

# Prolog answer computation

- Prolog searches the knowledge base for a fact that matches the query,
- when a match is found, it is marked.

# Prolog answer computation

- Prolog searches the knowledge base for a fact that matches the query,
- when a match is found, it is marked.
- if the user presses "`Enter`", the search is over,

# Prolog answer computation

- Prolog searches the knowledge base for a fact that matches the query,
- when a match is found, it is marked.
- if the user presses "`Enter`", the search is over,
- if the user presses ";" then "`Enter`", Prolog looks for a new match, from the previously marked place, and with the variable(s) in the query uninstantiated.

# Prolog answer computation

- Prolog searches the knowledge base for a fact that matches the query,
- when a match is found, it is marked.
- if the user presses "`Enter`", the search is over,
- if the user presses ";" then "`Enter`", Prolog looks for a new match, from the previously marked place, and with the variable(s) in the query uninstantiated.
- In the example above, two more "`; Enter`" will determine Prolog to answer:

```
X = mary.
false
```

# Prolog answer computation

- Prolog searches the knowledge base for a fact that matches the query,
- when a match is found, it is marked.
- if the user presses "`Enter`", the search is over,
- if the user presses ";" then "`Enter`", Prolog looks for a new match, from the previously marked place, and with the variable(s) in the query uninstantiated.
- In the example above, two more "`; Enter`" will determine Prolog to answer:

```
X = mary.
false
```

- When no (more) matching facts are found in the knowledge base, Prolog answers `false`.

# Conjunctions: more complex queries

- Consider the following facts:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

# Conjunctions: more complex queries

- Consider the following facts:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

- And the query:

```
?- likes(john, mary), likes(mary, john).
```

# Conjunctions: more complex queries

- Consider the following facts:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

- And the query:

```
?- likes(john, mary), likes(mary, john).
```

  - the query reads “does john like mary and does mary like john?”

# Conjunctions: more complex queries

- Consider the following facts:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

- And the query:

```
?- likes(john, mary), likes(mary, john).
```

  - the query reads "does john like mary and does mary like john?"
  - Prolog will answer `false`: it searches for each goal in turn (all goals have to be satisfied, if not, it will fail, i.e. answer `false`).

# Conjunctions: more complex queries

- For the query:

```
?- likes(mary, X), likes(john, X).
```

# Conjunctions: more complex queries

- For the query:

```
?- likes(mary, X), likes(john, X).
```

  - Prolog: try to satisfy the first goal (if it is satisfied put a placemarker), then try to satisfy the second goal (if yes, put a placemarker).

# Conjunctions: more complex queries

- For the query:

```
?- likes(mary, X), likes(john, X).
```

  - Prolog: try to satisfy the first goal (if it is satisfied put a placemarker), then try to satisfy the second goal (if yes, put a placemarker).
  - If at any point there is a failure, backtrack to the last placemarker and try alternatives.

# Example: conjunction, backtracking

The way Prolog computes the answer to the above query is represented:

- In Figure 1, the first goal is satisfied, Prolog attempts to find a match for the second goal (with the variable instantiated).
- The failure to find a match in the knowledge base causes backtracking, see Figure 2.
- The new alternative tried is successful for both goals, see Figure 3.

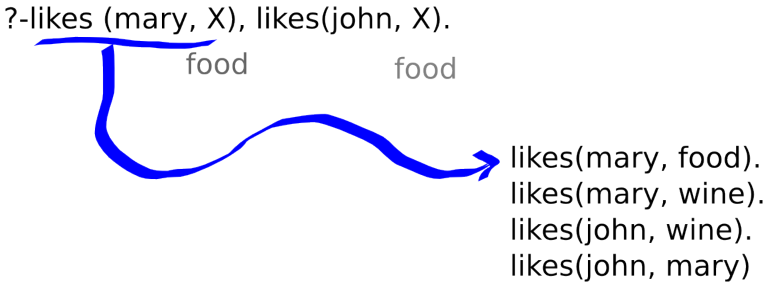


Figure: Success for the first goal.

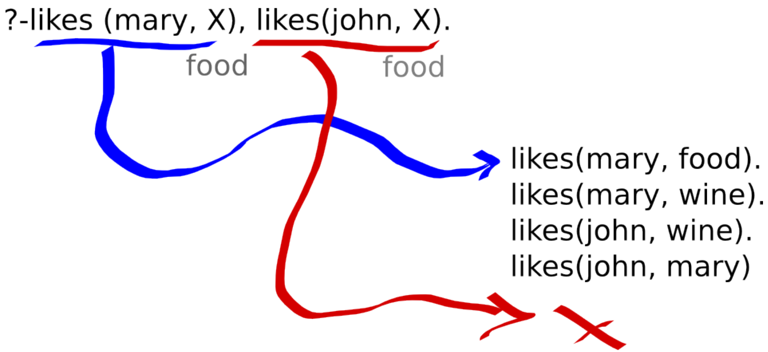


Figure: Second goal failure causes backtracking.

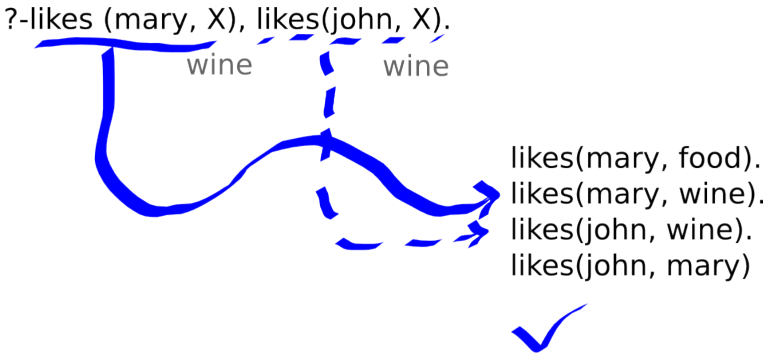


Figure: Success with alternative instantiation.

# Rules

- “John likes all people” can be represented as:

# Rules

- "John likes all people" can be represented as:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

but this is tedious!!!

# Rules

- "John likes all people" can be represented as:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

but this is tedious!!!

```
likes(john, X).
```

but this should be only for people!!!

# Rules

- "John likes all people" can be represented as:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

  but this is tedious!!!

```
likes(john, X).
```

  but this should be only for people!!!

- Enter rules: "John likes any object, but only that which is a person" is a rule about what (who) John likes.

# Rules

- "John likes all people" can be represented as:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

  but this is tedious!!!

```
likes(john, X).
```

  but this should be only for people!!!

- Enter rules: "John likes any object, but only that which is a person" is a rule about what (who) John likes.
- Rules express that a fact depends on other facts.

# Rules as definitions

- Rules can be used to express "definitions".

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."

- Example:

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."

- Example:

  "X is a sister of Y if X is female and X and Y have the same parents."

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."
- Example:

  "X is a sister of Y if X is female and X and Y have the same parents."
- Attention! The above notion of "definition" is not the same as the notion of definition in logic:

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."
- Example:

  "X is a sister of Y if X is female and X and Y have the same parents."
- Attention! The above notion of "definition" is not the same as the notion of definition in logic:
  - such definitions allow detection of the predicates in the head of the rule,

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."
- Example:

  "X is a sister of Y if X is female and X and Y have the same parents."
- Attention! The above notion of "definition" is not the same as the notion of definition in logic:
  - such definitions allow detection of the predicates in the head of the rule,
  - but there may be other ways (i.e. other rules with the same head) to detect such predicates,

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."
- Example:

  "X is a sister of Y if X is female and X and Y have the same parents."
- Attention! The above notion of "definition" is not the same as the notion of definition in logic:
  - such definitions allow detection of the predicates in the head of the rule,
  - but there may be other ways (i.e. other rules with the same head) to detect such predicates,
  - in order to have full definitions "iff" is needed instead of "if".

# Rules as definitions

- Rules can be used to express "definitions".
- Example:

  "X is a bird if X is an animal and X has feathers."
- Example:

  "X is a sister of Y if X is female and X and Y have the same parents."
- Attention! The above notion of "definition" is not the same as the notion of definition in logic:
  - such definitions allow detection of the predicates in the head of the rule,
  - but there may be other ways (i.e. other rules with the same head) to detect such predicates,
  - in order to have full definitions "iff" is needed instead of "if".
- Rules are general statements about objects and their relationships (in general variables occur in rules, but not always).

# Rules in Prolog

- Rules in Prolog have a head and a body.

# Rules in Prolog

- Rules in Prolog have a head and a body.
- The body of the rule describes the goals that have to be satisfied for the head to be true.

# Rules in Prolog

- Rules in Prolog have a head and a body.
- The body of the rule describes the goals that have to be satisfied for the head to be true.
- Example:

```
likes(john, X) :-
        likes(X, wine).
likes(john, X) :-
        likes(X, wine), likes(X, food).
likes(john, X) :-
        female(X), likes(X, wine).
```

# Rules in Prolog

- Rules in Prolog have a head and a body.
- The body of the rule describes the goals that have to be satisfied for the head to be true.
- Example:

```
likes(john, X) :-
        likes(X, wine).
likes(john, X) :-
        likes(X, wine), likes(X, food).
likes(john, X) :-
        female(X), likes(X, wine).
```

- Attention! The scope of the variables that occur in a rule is the rule itself (rules do not share variables).

# Example (royals)

- Knowledge base:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(alice, albert, victoria).
parents(edward, albert, victoria).
sister_of(X, Y):-
            female(X),
            parents(X, M, F).
            parents(Y, M, F).
```

# Example (royals)

- Knowledge base:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(alice, albert, victoria).
parents(edward, albert, victoria).
sister_of(X, Y):-
             female(X),
             parents(X, M, F).
             parents(Y, M, F).
```

- Goals:

```
?-sister_of(alice, edward).
?-sister_of(alice, X).
```

# Exercise (thieves)

- Consider the following:

```
/*1*/  thief(john).

/*2*/  likes(mary, food).
/*3*/  likes(mary, wine).
/*4*/  likes(john, X):- likes(X, wine).

/*5*/  may_steal(X, Y) :-
              thief(X), likes(X, Y).
```

- Explain how the query

```
?-may_steal(john, X).
```

is executed by Prolog.

- Prolog programs are built from terms (written as strings of characters).
- The following are terms:
  - constants,
  - variables,
  - structures.

# Constants

- Constants are simple (basic) terms.

# Constants

- Constants are simple (basic) terms.
- They name specific things or predicates (no functions in Prolog).

# Constants

- Constants are simple (basic) terms.
- They name specific things or predicates (no functions in Prolog).
- Constants are of 2 types:

# Constants

- Constants are simple (basic) terms.
- They name specific things or predicates (no functions in Prolog).
- Constants are of 2 types:
  - atoms,

# Constants

- Constants are simple (basic) terms.
- They name specific things or predicates (no functions in Prolog).
- Constants are of 2 types:
  - atoms,
  - numbers: integers, rationals (with special libraries), reals (floating point representation).

# Examples of atoms

- atoms:
  - likes ,
  - a (lowercase letters),
  - $=$,
  - $-->$,
  - 'Void' (anything between single quotes),
  - george_smith (constants may contain underscore),
- not atoms:
  - 314a5 (cannot start with a number),
  - george$-$smith (cannot contain a hyphen),
  - George (cannot start with a capital letter),
  - _something (cannot start with underscore).

# Variables

- Variables are simple (basic) terms,
- written in uppercase or starting with underscore _,
- Examples: X, Input, _something, _ (the last one called anonymous variable).
- Anonymous variables need not have consistent interpretations (they need not be bound to the same value):

```
?- likes(_,john).  % does anybody like John?
?- likes(_,_).     % does anybody like anybody?
```

# Structures

- Structure are compound terms, single objects consisting of collections of objects (terms),
- they are used to organize the data.
- A structure is specified by its functor (name) and its components

```
owns(john,book(wuthering_heights,bronte)).
book(wuthering_heights,author(emily,bronte)).

?-owns(john,book(X,author(Y,bronte))).
% does John own a book (X) by Bronte (Y,bronte)?
```

# Characters in Prolog

- Characters:
  - A-Z
  - a-z
  - 0-9
  - + - * / \ ~ ^ < > : .
  - ? @ # $ &
- Characters are ASCII (printing) characters with codes greater than 32.
- Remark: ' ' allows the use of any character.

# Arithmetic operators

- Arithmetic operators:
  - $+$,
  - $-$,
  - $*$,
  - $/$,
- $+(x, *(y, z))$ is equivalent with $x + (y \cdot z)$
- Operators do not cause evaluation in Prolog.
- Example: 3+4 (structure) does not have the same meaning with 7 (term).
- X is 3+4 causes evaluation ( is represents the evaluator in Prolog).
- The result of the evaluation is that X is assigned the value 7.

# Parsing arithmetic expressions

- To parse an arithmetic expression you need to know:
  - The position:
    - infix: $x + y$, $x * y$
    - prefix $-x$
    - postfix $x!$
  - Precedence: $x + y * z$ ?
  - Associativity: What is $x + y + z$? $x + (y + z)$ or $(x + y) + z$?

- Each operator has a precedence class:
    - 1 - highest
    - 2 - lower
    - ...
    - lowest
- $* \, /$ have higher precedence than $+ \, -$
- $8/2/2$ evaluates to:
    - $8$ $(8/(2/2))$ - right associative?
    - or $2$ $((8/2)/2)$ - left associative?
- Arithmetic operators are left associative.

# The unification predicate '='

- = - infix built-in predicate.

  $$?\!-X = Y.$$

  Prolog will try to match(unify) X and Y, and will answer true if successful.

# The unification predicate '='

- $=$ - infix built-in predicate.

$$?\!-X = Y.$$

Prolog will try to match(unify) X and Y, and will answer true if successful.

- In general, we try to unify 2 terms (which can be any of constants, variables, structures):

$$?\!-T1 = T2.$$

# The unification predicate '='

- $=$ - infix built-in predicate.

  ?–X = Y.

  Prolog will try to match(unify) X and Y, and will answer true if successful.

- In general, we try to unify 2 terms (which can be any of constants, variables, structures):

  ?–T1 = T2.

- Remark on terminology: while in some Prolog sources the term “matching” is used, note that in the (logic) literature matching is used for the situation where one of the terms is ground (i.e. contains no variables). What $=$ does is unification.

# The unification procedure

Summary of the unification procedure $?-\ T1 = T2$:

- If T1 and T2 are identical constants, success (Prolog answers true);

# The unification procedure

Summary of the unification procedure $?- \ T1 = T2$:

- If T1 and T2 are identical constants, success (Prolog answers true);
- If T1 and T2 are uninstantiated variable, success (variable renaming);

# The unification procedure

Summary of the unification procedure ?− T1 = T2:

- If T1 and T2 are identical constants, success (Prolog answers true);
- If T1 and T2 are uninstantiated variable, success (variable renaming);
- If T1 is an uninstantiated variable and T2 is a constant or a structure, success, and T1 is instantiated with T2;

# The unification procedure

Summary of the unification procedure ?− T1 = T2:

- If T1 and T2 are identical constants, success (Prolog answers true);
- If T1 and T2 are uninstantiated variable, success (variable renaming);
- If T1 is an uninstantiated variable and T2 is a constant or a structure, success, and T1 is instantiated with T2;
- If T1 and T2 are instantiated variables, then decide according to their value (they unify - if they have the same value, otherwise not);

# The unification procedure

Summary of the unification procedure ?− T1 = T2:

- If T1 and T2 are identical constants, success (Prolog answers true);
- If T1 and T2 are uninstantiated variable, success (variable renaming);
- If T1 is an uninstantiated variable and T2 is a constant or a structure, success, and T1 is instantiated with T2;
- If T1 and T2 are instantiated variables, then decide according to their value (they unify - if they have the same value, otherwise not);
- If T1 is a structure: $f(X_1, X_2, ..., X_n)$ and T2 has the same functor (name): $f(Y_1, Y_2, ..., Y_n)$ and the same number of arguments, then unify these arguments recursively ($X_1 = Y_1$, $X_2 = Y_2$, etc.). If all the arguments unify, then the answer is true, otherwise the answer is false (unification fails);

# The unification procedure

Summary of the unification procedure ?− T1 = T2:

- If T1 and T2 are identical constants, success (Prolog answers true);
- If T1 and T2 are uninstantiated variable, success (variable renaming);
- If T1 is an uninstantiated variable and T2 is a constant or a structure, success, and T1 is instantiated with T2;
- If T1 and T2 are instantiated variables, then decide according to their value (they unify - if they have the same value, otherwise not);
- If T1 is a structure: $f(X_1, X_2, ..., X_n)$ and T2 has the same functor (name): $f(Y_1, Y_2, ..., Y_n)$ and the same number of arguments, then unify these arguments recursively ($X_1 = Y_1$, $X_2 = Y_2$, etc.). If all the arguments unify, then the answer is true, otherwise the answer is false (unification fails);
- In any other case, unification fails.

# Occurs check

- Consider the following unification problem:

```
?- X = f(X).
```

# Occurs check

- Consider the following unification problem:

```
?- X = f(X).
```

- Answer of Prolog:

```
X = f(**).
```

# Occurs check

- Consider the following unification problem:

```
?- X = f(X).
```

- Answer of Prolog:

```
X = f(**).

X = f(X).
```

# Occurs check

- Consider the following unification problem:

```
?- X = f(X).
```

- Answer of Prolog:

```
X = f(**).

X = f(X).
```

- In fact this is due to the fact that according to the unification procedure, the result is $X = f(X) = f(f(X)) = ... = f(f(...(f(X)...)))$ - an infinite loop would be generated.

- Unification should fail in such situations.

- Unification should fail in such situations.
- To avoid them, perform an occurs check: If T1 is a variable and T2 a structure, in an expression like T1 = T2 make sure that T1 does not occur in T2.

- Unification should fail in such situations.
- To avoid them, perform an occurs check: If T1 is a variable and T2 a structure, in an expression like T1 $=$ T2 make sure that T1 does not occur in T2.
- Occurence check is deactivated by default in most Prolog implementations (is computationally very expensive) - Prolog trades correctness for speed.

- Unification should fail in such situations.
- To avoid them, perform an occurs check: If T1 is a variable and T2 a structure, in an expression like T1 = T2 make sure that T1 does not occur in T2.
- Occurence check is deactivated by default in most Prolog implementations (is computationally very expensive) - Prolog trades correctness for speed.
- A predicate complementary to unification:

- Unification should fail in such situations.
- To avoid them, perform an occurs check: If T1 is a variable and T2 a structure, in an expression like T1 = T2 make sure that T1 does not occur in T2.
- Occurence check is deactivated by default in most Prolog implementations (is computationally very expensive) - Prolog trades correctness for speed.
- A predicate complementary to unification:
  - \= succeeds only when = fails,

- Unification should fail in such situations.
- To avoid them, perform an occurs check: If T1 is a variable and T2 a structure, in an expression like T1 = T2 make sure that T1 does not occur in T2.
- Occurence check is deactivated by default in most Prolog implementations (is computationally very expensive) - Prolog trades correctness for speed.
- A predicate complementary to unification:
  - \= succeeds only when = fails,
  - i.e. T1 \= T2 cannot be unified.

# Built-in predicates for arithmetic

- Prolog has built-in numbers.
- Built-in predicates on numbers include:

  $X = Y$,
  $X \backslash= Y$,
  $X < Y$,
  $X > Y$,
  $X =< Y$,
  $X >= Y$,

  with the expected behaviour.
- Note that variables have to be instantiated in most cases (with the exception of the first two above, where unification is performed in the case of uninstantiation).

# The arithmetic evaluator is

- Prolog also provides arithmetic operators (functions), e.g.: $+$, $-$, $*$, $/$, mod, rem, abs, max, min, random, floor , ceiling etc, but these cannot be used directly for computation(2+3 means 2+3, not 5) - expressions involving operators are not evaluated by default.

# The arithmetic evaluator is

- Prolog also provides arithmetic operators (functions), e.g.: $+, -, *, /,$ mod, rem, abs, max, min, random, floor, ceiling etc, but these cannot be used directly for computation(2+3 means 2+3, not 5) - expressions involving operators are not evaluated by default.
- The Prolog evaluator is has the form:

```
X is Expr.
```

where X is an uninstantiated variable, and Expr is an arithmetic expression, where all variables must be instantiated (Prolog has no equation solver).

# Example (with arithmetic(1))

```
reigns(rhondri, 844, 878).
reigns(anarawd, 878, 916).
reigns(hywel_dda, 916, 950).
reigns(lago_ap_idwal, 950, 979).
reigns(hywel_ap_ieuaf, 979, 985).
reigns(cadwallon, 985, 986).
reigns(maredudd, 986, 999).
prince(X, Y):-
    reigns(X, A, B),
    Y >= A,
    Y =< B.

?- prince(cadwallon, 986).
true
?- prince(X, 979).
X = lago_ap_idwal ;
X = hywel_ap_ieuaf
```

# Example (with arithmetic(2))

```
pop(place1, 203).
pop(place2, 548).
pop(place3, 800).
pop(place4, 108).

area(place1, 3).
area(place2, 1).
area(place3, 4).
area(place4, 3).
density(X, Y):-
        pop(X, P),
        area(X, A),
        Y is P/A.

?-density(place3, X).
X = 200
true
```

- In this lecture the following were discussed:
  - asserting facts about objects,
  - asking questions about facts,
  - using variables, scopes of variables,
  - conjunctions,
  - an introduction to backtracking (in examples),
  - Prolog syntax: terms (constants, variables, structures),
  - Arithmetic in Prolog,
  - Unification procedure,
  - Subtle point: occurs check.

- All things SWIProlog can be found at `http://www.swi-prolog.org`.
- Install SWI-Prolog and try out the examples in the lecture.
- Read: Chapter 1 and Chapter 2 (including exercises section) of [Clocksin and Mellish, 2003].

Ben-Ari, M. (2001).
*Mathematical Logic for Computer Science*.
Springer Verlag, London, 2nd edition.

Clocksin, W. F. and Mellish, C. S. (2003).
*Programming in Prolog*.
Springer, Berlin, 5th edition.

Nilsson, U. and Maluszynski, J. (2000).
*Logic, Programming and Prolog*.
copyright Ulf Nilsson and Jan Maluszynski, 2nd edition.