Note: These lecture notes will be accompanied by additional explanations, demonstrations, and small-group exercises in class.

Course Content

- **Introduction to Prolog and Logic Programming.**
- **Prolog basic constructs:** Facts, rules, knowledge base, goals, unification, instantiation.
- **Prolog syntax, characters, equality and arithmetic.**
- **Data Structures:** Structures and trees, lists, strings.
- **Control Structures:**
  - Backtracking, recursion, cut and failure.
- **Input and output, assertions, consulting.**
- **Applications:** Databases, Artificial Intelligence
  - Games, natural language processing, meta-interpreters.

Prolog

- **Prolog = Programming in Logic.**
- Prolog is based on first order logic.
- Prolog is **declarative** (as opposed to **imperative**):
  - You specify what the problem is rather than how to solve it.
- Prolog is very useful in some areas (AI, natural language processing), but less useful in others (graphics, numerical algorithms).

Propositional Logic

- **Propositions** are statements that can be assigned a truth value
  - Elephants are pink. true or false?
- **Operators** for assigning truth values to combinations of propositions (sentences)

<table>
<thead>
<tr>
<th>Symbolic statement</th>
<th>Translation</th>
<th>Informal characterization</th>
</tr>
</thead>
<tbody>
<tr>
<td>p∧q</td>
<td>p and q</td>
<td>p ∧ q is true when both p and q are true</td>
</tr>
<tr>
<td>p∨q</td>
<td>p or q</td>
<td>p ∨ q is true when either p or q or both p and q are true</td>
</tr>
<tr>
<td>p⇒q</td>
<td>p logically implies q</td>
<td>p ⇒ q is true when p and q are both true, or p is false</td>
</tr>
<tr>
<td>p⇔q</td>
<td>p is logically equivalent to q</td>
<td>p ⇔ q is true if p and q have the same truth value</td>
</tr>
<tr>
<td>¬p</td>
<td>not p</td>
<td>¬p is true when p is false</td>
</tr>
</tbody>
</table>

Predicate Logic

- Involves entities and relations between entities.
- **Entities** are expressed using:
  - **Variables:** X, Y, Somebody, Anybody
  - **Constants:** fido, fiffy, bigger, dog, has, bone
- **Logical operators** - connectors between relations and (¬), or(∨), not(¬), logically implies (⇒), logically equivalent (⇔), for all (∀), exists (∃)
- **Relations** are expressed using:
  - **Predicates** - express a simple relation among entities, or a property of some entity
    - fido is a dog - dog(fido)
    - fiffy is a dog - dog(fiffy)
    - fido is bigger than fiffy - bigger(fido, fiffy)
- Formulas - express a more complex relation among entities
  - if fido is bigger than fiffy, and fiffy has a bone, then fido can take the bone
    (bigger(fido, fiffy) ∧ has(fiffy, bone)) ⇒ can_take(fido, bone)
- Sentences - are formulas with no free variables
  - dog(X) contains a variable which is said to be free while the X in ∀ X.dog(X) is bound.
Logic → Prolog

- Involves entities and relations between entities.
- Entities are expressed using:
  - Variables: X, Y, Somebody, Anybody
  - Constants: fido, fiffy, bigger, dog, has, bone
- Logical operators: connectors between relations
  - and (,), or (;), not (\+), is logically implied by (:-)
- Relations are expressed using:
  - Predicates - relation among entities, or a property of an entity
  - fido is a dog - dog(fido)
  - fido is bigger than fiffy - bigger(fido, fiffy)

Logic → Prolog (cont.)

- Rules - complex relation among entities
  - if fido is bigger than fiffy, and fiffy has a bone, then fido can take the bone
    can_take(fido, bone) :-
    bigger(fido, fiffy), has(fiffy, bone).
  - Or more general:
    can_take(Dog1, bone) :-
    bigger(Dog1, Dog2),
    has(Dog2, bone).

Programming Language Comparison

Imperative programming languages

- "procedural" -> they describe how a sequence of instructions compute the result to a certain problem.
- we concentrate on how to formulate a solution in terms of the primitive operations available
- what you see is what is being done

Logic programming languages

- specify the problem in a declarative style (facts about objects, relations between objects), describe what is the objective and let the system prove it
- we concentrate on problem
- there are underlying mechanisms that help the program reach its goal

Logic Programming

- A program in logic is a definition (declaration) of the world - the entities and the relations between them.
- Logic programs establishing a theorem (goal) and asks the system to prove it.
- Satisfying the goal:
  - yes, prove the goal using the information from the knowledge base
  - no:
  - cannot prove the truth of the goal using the information from the knowledge base
  - the goal is false according to available information

Definitions

- Three basic constructs in Prolog
  - Facts, rules, and queries.
- Knowledge base (database)
  - A collection of facts and rules.
  - Prolog programs are knowledge bases.
- We use Prolog programs by posing queries.

Facts

- Facts are used to state things that are unconditionally true.
  - We pay taxes. we_pay_taxes.
  - The earth is round. The sky is blue.
    round(earth).
    blue(sky).
  - Beethoven was a composer that lived between 1770 and 1827.
    composer(beethoven,1770,1827).
  - Tom is the parent of Liz.
    parent(liz, tom).
  - fido is bigger than fiffy.
    bigger(fido, fiffy).

Exercise:
SWI-Prolog

- The SWI department of the University of Amsterdam.
- Free
- Small
- Available in the lab
- Download a copy to work at home
  http://www/swi-prolog.org
- Documentation

Queries on Facts

- John likes apples, csi2165 and Mary.
  likes(john, apples).
  likes(john, csi2165).
  likes(john, mary).
- Does John like apples?
  ?-likes(john, apples).
  yes
- What does John like?
  ?-likes(john, X).
  X = apples;
  X = csi2165;
  X = mary;
  no

For more solutions, ‘Enter’ to stop.

Building a Knowledge Base (Lab 1)

Problem 1. Build a knowledge base to represent the parent relationships that can be deduced from the tree
parent(pam, bob).

Problem 2. Build predicates that describe the following family relationships:
grandparent/2
mother/2 father/2
brother/2 sister/2 sibling/2
aunt/2 uncle/2
precursor/2

Rules

- If there is smoke there is fire. 
  fire :- smoke.
- Liz is an offspring of Tom if Tom is a parent of Liz.
  offspring(liz, tom) :- parent(tom, liz).
- Y is an offspring of X if X is a parent of Y.
  offspring(Y, X) :- parent(X, Y).
- Two persons are sisters if they are females and have the same parents.
  siblings(P1, P2) :- parent(P, P1), parent(P, P2).

Exercise:
- Family relations
  grandparent(X, Y) :-

Queries on Rules

- Mary drinks beer. Terry drinks beer.
  drinks(mary, beer).
  drinks(terry, beer).
- John likes everybody who drinks beer.
  likes(john, X) :- drinks(x, beer).
- Does John like Mary?
  ?-likes(john, mary).
  yes
- Who does John like?
  ?-likes(john, X).
  X = mary;
  X = terry;
  no

Clauses

- In Prolog, rules (and facts) are called clauses.
- A clause always ends with ‘.’
- Clause:
  <head> :- <body>.
  – you can conclude that <head> is true, if you can prove that <body> is true.
- Facts - clauses with an empty body: <head>.
  – you can conclude that <head> is true.
- Rules - normal clauses (or more clauses)
- Queries - clauses with an empty head: ?- <body>.
  – Try to prove that <body> is true.
**Rules**

- Rules state information that is conditionally true of the domain of interest.
- The general form of these properties
  - \( p \) is true if \( (p_1 \text{ is true, and } p_2 \text{ is true, } \ldots \text{ and } p_n \text{ is true}) \)
- Horn clause
  - \( P :- p_1, p_2, \ldots, p_n \).
- Interpretation (Prolog):
  - in order to prove that \( p \) is true, the interpreter will prove that each of \( p_1, p_2, \ldots, p_n \) is true
  - \( p \) - the head of the rule
  - \( p_1, p_2, \ldots, p_n \) - the body of the rule (subgoals)

**Rules and Conjunctions**

- A man is happy if he is rich and famous.
- In Prolog:
  
  \[
  \text{happy(Person)} :- \text{man(Person),}
  \text{rich(Person),}
  \text{famous(Person)}.
  \]

**Rules and Disjunctions**

- Someone is happy if he/she is healthy, wealthy or wise.
- In Prolog:
  
  \[
  \text{happy(Person)} :- \text{healthy(Person).}
  \text{happy(Person)} :- \text{wealthy(Person).}
  \text{happy(Person)} :- \text{wise(Person).}
  \]

**Both Disjunctions and Conjunctions**

- A woman is happy if she is healthy, wealthy or wise.
- In Prolog:
  
  \[
  \text{happy(Person)} :- \text{healthy(Person), woman(Person).}
  \text{happy(Person)} :- \text{wealthy(Person), woman(Person).}
  \text{happy(Person)} :- \text{wise(Person), woman(Person).}
  \]

**Variables**

- Objects referred by a name starting with a capital letter.
- Scope rule:
  - Two uses of an identical name for a logical variable only refer to the same object if the uses are within a single clause.

**Queries**

- The goal represented as a question.
  
  \[
  ?- \text{round(earth).} /* \text{is it true that the earth is round?} */
  \]
  
  \[
  ?- \text{round(X).} /* \text{is it true that there are entities which are round? (what entities are round?)} */
  \]
  
  \[
  ?- \text{composer(beethoven, 1770, 1827).} /* \text{is it true that Beethoven was a composer who lived between 1770 and 1827?} */
  \]
  
  \[
  ?- \text{owns(john, book).} /* \text{is it true that john owns a book?} */
  \]
  
  \[
  ?- \text{owns(john, X).} /* \text{is it true that john owns something?} */
  \]
Predicate

• composer(beethoven, 1770, 1827) → predicate
• composer → functor
• beethoven, 1770, 1827 → arguments
• number of arguments: 3 → arity.
• write as composer/3

Example

We have a Prolog program:

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

Now we pose the query:

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

What answers do we get?
Declarative Semantics (what)

- **Declarative semantics** - telling Prolog what we know.
- If we don’t know if something is true, we assume it is false - closed world assumption.
- Sometimes we tell it relations that we know are false. (sometimes it is easier to show that the opposite of a relation is false, than to show that the relation is true)

I know (it is true) that the max between two numbers X and Y is X, if X is bigger than Y.  \[ \text{max}(X, Y, X) :- X > Y. \]

I know that the max between two numbers X and Y is Y if Y is bigger or equal to X.  \[ \text{max}(X, Y, Y) :- Y \geq X. \]

?- \text{max}(1, 2, X).

Declarative Semantics (cont.)

I know that 0 is a positive integer.
\[ \text{positive_integer}(0). \]

I know that X is a positive integer if there is another positive integer Y such that X is Y+1.
\[ \text{positive_integer}(X) :- \text{positive_integer}(Y), X \geq Y+1. \]

?- positive_integer(3).

?- positive_integer(X).

Procedural Semantics (how)

- **Procedural semantics** - how do I prove a goal?

\[ \text{max}(X, Y, X) :- X > Y. \]

\[ \text{max}(X, Y, Y) :- Y \geq X. \]

?- \text{max}(1, 2, X).

If I can prove that X is bigger then Y, then I can prove that the max between X and Y is X.  
or, if that doesn’t work,  
If I can prove that Y is bigger or equal to X, then I can prove that the max between X and Y is Y.

Procedural Semantics (cont.)

\[ \text{positive_integer}(0). \]

\[ \text{positive_integer}(X) :- \text{positive_integer}(Y), X \geq Y+1. \]

?- positive_integer(3).

- If I can prove that X is 0, then I can prove that X is a positive integer  
or,  
- If I can prove that Y is a positive integer, and if X is Y+1, then I can prove that X is a positive integer.  
I can prove that Y is a positive integer if I can prove that Y is 0  
or  
If I can prove that Z is a positive integer, and if Y is Z+1, then I can prove ...

Terms

- Prolog programs are built from terms.
- Three types of terms
  - Constants
  - Variables
  - Structures
- Terms are composed of letters, digits and/or sign characters.

Another view: Objects

- Simple objects:
  - constants: for specific objects or specific relationships.
  - numbers (integers, floating point numbers)
  - atoms (bob, hello, *, '&%?', 'I'm not a variable')
- variables:
  - anonymous variables
  - named variables
- Complex objects
  - lists
  - other structures
Variables

- Names that stand for objects that may already or may not yet be determined by a Prolog program
  - if the object a variable stands for is already determined, the variable is *instantiated*
  - if the object a variable stands for is not yet determined, the variable is *uninstantiated*
- A Prolog variable does not represent a location that contains a modifiable value; it behaves more like a mathematical variable (and has the same scope)
- An instantiated variable in Prolog cannot change its value

Variables (cont.)

- Constants in Prolog: numbers, strings that start with lowercase, anything between single quotes
- Variables in Prolog: names that start with an uppercase letter or with '_'
- Examples:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Var, Const</td>
<td>x, y, var, const</td>
</tr>
<tr>
<td>_var, _const</td>
<td>some_Thing, 1, 4</td>
</tr>
<tr>
<td>_</td>
<td>&quot;String&quot;, &quot;List of ASCII codes&quot;</td>
</tr>
</tbody>
</table>

Anonymous variables

- a variable that stands in for some unknown object
- stands for some objects about which we don’t care
- several anonymous variables in the same clause need not be given consistent interpretation
- written as _ in Prolog

?!- composer(X, _, _).
X = beethoven;
X = mozart;
...

We are interested in the names of composers but not their birth and death years.

Verify Type of a Term

- `var(+Term)`
  - Succeeds if `Term` is currently a free variable.
- `nonvar(+Term)`
  - Succeeds if `Term` is currently not a free variable.
- `integer(+Term)`
  - Succeeds if `Term` is bound to an integer.
- `float(+Term)`
  - Succeeds if `Term` is bound to a floating point number.
- `number(+Term)`
  - Succeeds if `Term` is bound to an integer or a floating point number.
- `atom(+Term)`
  - Succeeds if `Term` is bound to an atom.
- `string(+Term)`
  - Succeeds if `Term` is bound to a string.
- `atomic(+Term)`
  - Succeeds if `Term` is bound to an atom, string, integer or float.
- `compound(+Term)`
  - Succeeds if `Term` is bound to a compound term.

Some Built-in Predicates (Operators)

<table>
<thead>
<tr>
<th>for constants:</th>
<th>for variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>number/1</td>
<td>var/1</td>
</tr>
<tr>
<td>integer/1</td>
<td>nonvar/1</td>
</tr>
<tr>
<td>float/1</td>
<td>is/2</td>
</tr>
<tr>
<td>atom/1</td>
<td>=/2</td>
</tr>
<tr>
<td>atomic/1</td>
<td></td>
</tr>
</tbody>
</table>

Examples:

number(15) atom(my_atom)
number(0.001) atom('This?')
integer(16) atom(15)
integer(1.0) atomic(a)
float(1.5E-1) atomic(4)
float(1.0) atomic(4.2E+01)

Structures

- Structures are objects that have several components, which in turn can be structures.
- Structures are treated in the program as single objects.
- `functor` is used to combine components into a single object.
- A `functor` must be an atom.
- For example:
  - `date(1, may,1999)`
  - `course(csc 2165, fall2005)`
Structure: Example

- Description:
  - a person has:
    - name - first name, last name
    - birth date - day, month, year
    - occupation
- Prolog representation - example
  - person(name(michael, jordan), birth_date(17, february, 1963), occupation('NBA player'))

Data Objects in Prolog (Summary)

- Structures
- Simple objects
  - Constants
  - Variables
    - Atoms
    - Numbers
    - Named variables
    - Anonymous variables

Structures - Exercise

- Description:
  - point in the 2D space
  - triangle
  - a country
    - has a name
    - is located in a continent at a certain position
    - has population
    - has capital city which has a certain population
    - has area

Structures - Exercise

- Knowledge base:
  - country(canada, location(america, north), population(30), capital('Ottawa',1),area(_)).
  - country(usa, location(america, north), population(200),capital('Washington DC', 17), area(_)).

A Particular Structure

How can we represent the courses a student takes?
- courses(cs2111, cs2114, cs2165)
- courses(cs2114, cs2115, cs2165, mat2343)
- courses(adm2302, cs2111, cs2114, cs2115, cs2165)

Three different structures.

In general, how do we represent a variable number of arguments with a single structure?

A Particular Structure

Consider a single structure courses/2:
- the first argument - a course
- the second argument - a courses/2 structure
- courses(cs2111, courses(cs2114, courses(cs2165, nil)))
- courses(cs2114, courses(cs2165, nil))

They are lists!

That's useful but too messy, better use lists:
- [cs2111, cs2114, cs2165]
- [cs2114, cs2165]
Lists

- **Functor name**: .
- **Arity**: 2
  - First argument - can be anything - called the **head** of the list
  - Second argument - must be a list - called the **tail** of the list
- **Representing the lists**: .(Head1, Tail) = [Head1 | Tail] = [Head1, HeadOfTail | TailOfTail] = … = [Head1, Head2, Head3, ..., LastHead | []] = [Head1, Head2, …, LastHead]
  (“…” here is not a Prolog notation)
- We use the square bracket notation in our program since it is more readable.

- **Examples**: 
  - [a,b,c] = .(a, .(b, .(c, []))) = [a | [b | [c | []]]] = [a, b, c | []]
  - [1,2,3] = .(1, .(2, .(3, []))) = [1 | [2 | [3 | []]]]

- **Exercise**:
  - | List | Head | Tail |
    |--------|------|------|
    | [a, b, c] | a | [b, c] |
    | [1, 2, 3] | 1 | [2, 3] |

Try them out in SWI-Prolog

Matching, Unification, and Instantiation

- Prolog will try to find in the knowledge base a fact or a rule which can be used in order to prove a goal
- **Proving**:
  - match: the goal on a fact or head of some rule. If matching succeeds, then:
  - unify: the goal with the fact or the head of the rule. As a result of unification:
  - instantiate: the variables (if there are any), such that the matching succeeds
- NB: variables in Prolog cannot change their value once they are instantiated!

Matching

- **Matching**: Prolog tries to find a fact or a head of some rule with which to match the current goal
- **Match**: the functor and the arguments of the current goal, with the functor and the arguments of the fact or head of rule
- **Rules for matching**:
  - constants only match an identical constant
  - variables can match anything, including other variables

<table>
<thead>
<tr>
<th>Goal</th>
<th>Predicate</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>constant</td>
<td>yes</td>
</tr>
<tr>
<td>constant</td>
<td>other_constant</td>
<td>no</td>
</tr>
<tr>
<td>Var</td>
<td>some_constant</td>
<td>yes</td>
</tr>
<tr>
<td>Var</td>
<td>Other_Var</td>
<td>yes</td>
</tr>
<tr>
<td>some_constant</td>
<td>Some_Var</td>
<td>yes</td>
</tr>
</tbody>
</table>

Unification

- Done after a match between the current goal and a fact or the head of a rule is found
- It attaches values to variables (**instantiates** the variables), such that the goal and the predicate are a perfect match:
  - match:
    - goal - composer(beethoven, B, D) with fact - composer(beethoven,1770,1827)
  - unification:
    - B will be instantiated to 1770
    - D will be instantiated to 1827
Unification (cont.)

- If the match is done on the head of some rule, then the instantiations done for the variables are also valid in the body of the rule:
  - match:
    - goal - contemporaries(beethoven, mozart) with head of contemporaries(X, Y) :- composer(X, B1, D1), composer(Y, B2, D2), X \( \neq \) Y, ... 
  - unification:
    - X will be instantiated to beethoven
    - Y will be instantiated to mozart and now the rule will look: contemporaries(beethoven, mozart) :- composer(beethoven,B1,D1), composer(mozart, B2, D2), beethoven \( \neq \) mozart, ...

Instantiation and Unification - Exercise

<table>
<thead>
<tr>
<th>unify</th>
<th>with</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>likes(jim, piano)</td>
<td>likes(jim, X)</td>
<td></td>
</tr>
<tr>
<td>likes(jim, X)</td>
<td>likes(Y, piano)</td>
<td></td>
</tr>
<tr>
<td>owns(X, Y)</td>
<td>owns(jim, calliope)</td>
<td></td>
</tr>
<tr>
<td>owns(X, Y)</td>
<td>owns(Y, X)</td>
<td></td>
</tr>
<tr>
<td>owns(jim, piano)</td>
<td>likes(jim, piano)</td>
<td></td>
</tr>
<tr>
<td>owns(jim, piano)</td>
<td>owns(bill, piano)</td>
<td></td>
</tr>
<tr>
<td>owns(jim, X, Y)</td>
<td>owns(jim, piano)</td>
<td></td>
</tr>
</tbody>
</table>

Work them out and validate them with SWI-Prolog.

Structures Matching and Unification

- Matching on structures:
  - match the functor of the two structures
  - match each argument of the two structures (if some argument is complex, match it according to the same rules)

- Example:
  - a(b,c) a(b,c) \( \rightarrow \) match
  - a(b,C) a(b, x) \( \rightarrow \) C = x
  - a(X) a(B, c) \( \rightarrow \) don't match

Structure Matching and Unification - Exercise

<table>
<thead>
<tr>
<th>Structure 1 = Structure 2:</th>
<th>Instantiations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(b, X) = a(Y, c)</td>
<td>a(b, X) = a(Y, c)</td>
</tr>
<tr>
<td>a(b, X) = a(X, Y)</td>
<td>a(b, X) = a(X, Y)</td>
</tr>
<tr>
<td>a(b, X) = a(b, c(0))</td>
<td>a(b, X) = a(b, c(0))</td>
</tr>
<tr>
<td>a(b(X), Y) = a(Y, c(2))</td>
<td>a(b(X), Y) = a(Y, c(2))</td>
</tr>
<tr>
<td>a(b(X), Y) = a(b(Y), Z)</td>
<td>a(b(X), Y) = a(b(Y), Z)</td>
</tr>
<tr>
<td>X = [john,skates]</td>
<td>X = [X,Y,Z]</td>
</tr>
<tr>
<td>[cat] = [h</td>
<td>t]</td>
</tr>
<tr>
<td>[n(X,Y),a(U)] = [Name,Age]</td>
<td>X = a(b,c(0))</td>
</tr>
</tbody>
</table>

Work them out on your computer!

Structures - Another View

- We can view structures as trees:

```
person(name(michael,jordan),birth_date(17,february,1963),occupation('NBA player'))
```

![Diagram of structures as trees]
Three Kinds of Equality

• When are two terms said to be equal?
• We introduce 3 types of equality now (more later)
  – X = Y: this is true if X and Y match.
  – X = E: this is true if X matches the value of the arithmetic expression E.
  – T1 = T2: this is true if terms T1 and T2 are identical
    • Have exactly the same structure and all the corresponding components are the same. The name of the variables also have to be the same.
    • It is called: literal equality.
    • If X = Y, then X = Y. the former is a stricter form of equality.

Unification Operator: =

• = → unifies with: X = Y
  – succeeds as long as X and Y can be unified
  – X may or may not be instantiated
  – Y may or may not be instantiated
  – X and Y become bound together (they now refer to the same object)

  A = B = D, C = E = a, yes
• ? - a(b, X, c) = a(b, Y, c).
  X = Y, yes

Unification Operator: \=

• \= → does not unify with: X \= Y
  – succeeds as long as X and Y cannot be unified
  – both X and Y must be instantiated (why?)
  – X and Y may have uninstantiated elements inside them

• ? - [A, [B, C]] \= [A, B, C].
  yes
• ? - a(b, X, c) \= a(b, Y, c).
  no

Unification Operators: \==

• \== → not already instantiated to: X \== Y
  – succeeds as long as X and Y are not already instantiated to the same object

• ? - A \== hello.
  yes
• ? - a(b, X, c) \== a(b, Y, c).
  yes

Arithmetic Operator: is

• is → arithmetic evaluation : X is Expr
  – succeeds a long as X and the arithmetic evaluation of Expr can be unified
  – X may or may not be instantiated
  – Expr must not contain any uninstantiated variables
  – X is instantiated to the arithmetic evaluation of Expr

• ? - 5 is (3 * 7 + 1) / 4.
  X = 4
• ? - X is (3 * 4) +10 mod 6.
  X = 4

Question: would it make any difference if we replace = with \== in the siblings/2 on slide 14?
Summary of Part I

- Introduction to Prolog and Logic Programming.
- Prolog basic constructs: facts, rules, queries.
- Unification, variables.
- Prolog syntax, equality, and arithmetic.