# VNU-HUS MAT1206E/3508: Introduction to AI

# Logic Programming with PROLOG


Hoàng Anh Đức

Bộ môn Tin học, Khoa Toán-Cơ-Tin học
Đại học KHTN, ĐHQG Hà Nội
hoanganhduc@hus.edu.vn

# Contents

Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

# Additional Materials

Learn Prolog Now!

- `https://www.let.rug.nl/bos/lpn/index.php`
- by **Patrick Blackburn**, **Joost Bos**, and **Kristina Striegnitz**.



2 Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# PROLOG Systems and Implementations

- PROLOG = **Pro**gramming in **Log**ic
- PROLOG is used in many projects, primarily in AI and computational linguistics.
- We will now give a short introduction to this language, present the most important concepts, show its strengths, and compare it with other programming languages and theorem provers.
- Those looking for a complete programming course are directed to textbooks such as [Bratko 2011]; [Clocksin and Mellish 2013] and the documentations at `https://www.swi-prolog.org/` and `http://www.gprolog.org/`.
- PROLOG systems interpret *Warren Abstract Machine code (WAM)*.
- PROLOG source code is compiled into so-called WAM code, which is then interpreted by the WAM.
- **Performance:** up to 10 million logical inferences per second (LIPS) on a 1 Gigahertz PC



Additional Materials
3 PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# PROLOG Systems and Implementations

The syntax of the language PROLOG *only allows Horn clauses* (i.e., clauses having at most one positive literal)

| PL1 / clause normal form | PROLOG | Desc. |
|---|---|---|
| $(\neg A_1 \vee \cdots \vee \neg A_m \vee B)$ | `B:- A_1, ... , A_m.` | Rule |
| $(A_1 \wedge \cdots \wedge A_m) \Rightarrow B$ | `B:- A_1, ... , A_m.` | Rule |
| $A$ | `A.` | Fact |
| $(\neg A_1 \vee \cdots \vee \neg A_m)$ | `?- A_1, ... , A_m.` | Query |
| $\neg(A_1 \wedge \cdots \wedge A_m)$ | `?- A_1, ... , A_m.` | Query |

- Here $A_1, \ldots, A_m, A, B$ are literals.
- The literals are, as in PL1, constructed from predicate symbols with terms as arguments.
- As we can see in the above table, *in PROLOG there are no negations in the strict logical sense* because *the sign of a literal is determined by its position in the clause*.



Additional Materials
4 PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# PROLOG Systems and Implementations



Additional Materials

5 PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

A brief history

- Kowalski: late 60's Logician who showed logical proof can support computation.
- Colmerauer: early 70's Developed early version of PROLOG for natural language processing, mainly multiple parses.
- Warren: mid 70's First version of PROLOG that was efficient.
- Japan: early 80's The 5th Generation Computer Project chose to use PROLOG as the computer language for the AI programming.

# Basic of PROLOG

- A PROLOG program consists of *predicate definitions*.
- A predicate denotes a property or relationship between objects.
- Definitions consist of (Horn) *clauses*.
- A clause has a *head* and a *body* (*Rule*) or *just a head* (*Fact*).
- A head consists of a *predicate name* and *arguments*.
- A clause body consists of a conjunction of *terms*.
- Terms can be *constants*, *variables (with initial capital letter)*, or *compound terms*.
- We can set our program *goals* by typing a query, which is a list of atomic formulas.
- A goal unifies with clause heads in order (top down), and the body of the clause becomes new subgoals.
- *Unification* leads to the *instantiation* of variables to values.
- If any variables in the initial goal become instantiated this is reported back to the user.




Additional Materials

PROLOG Systems and Implementations

6 Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# Basic of PROLOG



## Variables

- *Variables* begin with a Capital letter, or "_"
  - For example, `X`, `Tom`, `_result`
- "_" is a nameless variable.
- A variable can have a value.

# Basic of PROLOG



## Atoms

- An *atom* is a constant in terms; it just stands for itself.
- Atoms do not begin with a capital letter
  - For example, `x`, `tom`
- Atomic formulas are called *structures in PROLOG*.
- You can make an atom containing any characters at all by enclosing it in *single quotes*:
  - For example, `'C:\\My Documents\\examples.pl'`
  - If you use double quotes, you will get a list of ASCII values, which is probably not what you want
    - `?- X = "Hello".` results `X = [72, 101, 108, 108, 111].`
  - In a quoted atom, a single quote must be doubled or backslashed
    - For example, `'Can''t, or won\'t?'`
  - Backslashes in file names must also be doubled
    - For example, `'C:\\My Documents\\examples.pl'`
    - Better yet, use forward slashes in paths; every OS, including Windows, understands this

# Basic of PROLOG

## Predicates

- A *predicate* is a definition of a *functor* (predicate symbol), which is *collection of clauses with the same functor and arity (number of arguments)*.
  - `loves(john, mary).`
  - `loves(mary, bill).`
  - `loves(chuck, X) :- female(X), rich(X).`
- These clauses should stay together.
- The scope of a variable (such as `X`) is the single clause in which it occurs.
- A PROLOG program is just a collection of predicates.

## Common Problems

- Capitalization is *meaningful*!
- No space is allowed between a functor and its argument list:
  - `man(tom)`, not `man (tom)`.
- Double quotes indicate a list of ASCII character values, *not* a string
- Don't forget the period! (But if you do, you can put it on the next line.)




Additional Materials

PROLOG Systems and Implementations

9 Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

## Central Ideas of PROLOG

- **SUCCESS (true) / FAILURE (false)**
  - any computation can “succeed” or “fail”, and this is used as a ‘test’ mechanism.
- **UNIFICATION (2-WAY MATCHING)**
  - any two data items can be compared for similarity, and values can be bound to variables in order to allow a match to succeed.
- **SEARCHING**
  - the whole activity of the PROLOG system is to search through various options to find a combination that succeeds.
- **BACKTRACKING**
  - when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.




Additional Materials

PROLOG Systems and Implementations

10 Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# Basic of PROLOG

## Running PROLOG Program

- Program is facts + rules. (horn clauses).
- Feed Query to PROLOG after "`?-`"
- A Query is a conjunct of atomic formulas $Q_1, Q_2, \ldots, Q_n$, written as ("`?-`" as "$\neg$")

$$\texttt{?-}\ Q_1\texttt{,}\ Q_2\texttt{,}\ \ldots\texttt{,}\ Q_n\texttt{.}$$

  It denotes $\neg Q_1 \vee \neg Q_2 \vee \cdots \vee \neg Q_n$, a negative clause.
- Substitutions for variables that solve the query are reported; if no variables, then PROLOG returns yes.
- Use "`;`" to get other solutions.

## Note

PROLOG programs can be prepared in a text file and loaded into PROLOG by

```
[filename].
```

or added on a terminal using `[user].` and `Ctrl-D`.



Additional Materials

PROLOG Systems and Implementations

11 Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# Simple Examples
## Family Relationships



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
12 Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

A brief recall from Example 5, Lecture “First-order Predicate Logic”

- *child*$(x, y, z)$ means “$x$ is a child of $y$ and $z$”
- *descendant*$(x, y)$ means “$x$ is a descendant of $y$”

# Simple Examples

## Family Relationships

The *knowledge base* $KB$ of family relationships

$$KB \equiv \textit{child}(\textit{oscar}, \textit{karen}, \textit{frank}) \wedge$$
$$\textit{child}(\textit{mary}, \textit{karen}, \textit{frank}) \wedge \textit{child}(\textit{eve}, \textit{anne}, \textit{oscar}) \wedge$$
$$\textit{child}(\textit{henry}, \textit{anne}, \textit{oscar}) \wedge \textit{child}(\textit{isabelle}, \textit{anne}, \textit{oscar}) \wedge$$
$$\textit{child}(\textit{clyde}, \textit{mary}, \textit{oscarb}) \wedge (\forall x \, \forall y \, \forall z \; \textit{child}(x, y, z) \Rightarrow \textit{child}(x, z, y))$$
$$\wedge \; (\forall x \, \forall y \; \textit{descendant}(x, y) \Leftrightarrow \exists z \; \textit{child}(x, y, z)$$
$$\vee \; (\exists u \, \exists v \; \textit{child}(x, u, v) \wedge \textit{descendant}(u, y)))$$

is coded as a PROLOG program

rel.pl

```prolog
child(oscar, karen, frank).
child(mary, karen, frank).
child(eve, anne, oscar).
child(henry, anne, oscar).
child(isolde, anne, oscar).
child(clyde, mary, oscarb).

child(X,Z,Y) :- child(X,Y,Z).

descendant(X,Y) :- child(X,Y,Z).
descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

```
?- [rel]. % load and compile rel.pl
true .

?- child(eve,oscar,anne). % initial query
true .
```



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
13 Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Simple Examples
## Family Relationships

How does the answer of the initial query come?

- For the query

```
?- child(eve,oscar,anne).
```

  there are six facts (lines 1-6) and one rule (line 8) with the same predicate in its clause head.
- Now unification is attempted between the query and each of the complementary literals in the input data in order of occurrence.
- If one of the alternatives fails, this results in backtracking to the last branching point, and the next alternative is tested.
- Because unification fails with every fact, the query is unified with the recursive rule in line 8. (`X/eve`, `Z/oscar`, `Y/anne`.)
- Now the system attempts to solve the subgoal `child(eve,anne,oscar)`, which succeeds with the third alternative.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
14 Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Simple Examples
## Family Relationships



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
15 Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

The next queries are answered with the first solutions found.

rel.pl

```
1  child(oscar, karen, frank).
2  child(mary, karen, frank).
3  child(eve, anne, oscar).
4  child(henry, anne, oscar).
5  child(isolde, anne, oscar).
6  child(clyde, mary, oscarb).
7
8  child(X,Z,Y) :- child(X,Y,Z).
9
10 descendant(X,Y) :- child(X,Y,Z).
11 descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

```
?- descendant(X,Y).
X = oscar,
Y = karen

?- descendant(clyde,Y).
Y = mary
```

However, the query

```
?- descendant(clyde,karen).
```

is not answered. This is because of the clause in line 8, which specifies symmetry of the child predicate. This clause calls itself recursively without the possibility of termination.

# Simple Examples

## Family Relationships

This problem can be solved with the following new program.

rel01.pl

```
child(oscar, karen, frank).
child(mary, karen, frank).
child(eve, anne, oscar).
child(henry, anne, oscar).
child(isolde, anne, oscar).
child(clyde, mary, oscarb).

descendant(X,Y) :- child(X,Y,Z).
descendant(X,Y) :- child(X,Z,Y).
descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

```
?- descendant(clyde, karen).
true .

?- child(eve,oscar,anne).
false .
```

But now the query

```
?- child(eve,oscar,anne).
```

is no longer correctly answered because the symmetry of `child` in the last two variables is no longer given.

# Simple Examples

## Family Relationships

A solution to both problems is found in the program.

rel02.pl

```
child_fact(oscar, karen, frank).
child_fact(mary, karen, frank).
child_fact(eve, anne, oscar).
child_fact(henry, anne, oscar).
child_fact(isolde, anne, oscar).
child_fact(clyde, mary, oscarb).

child(X,Z,Y) :- child_fact(X,Y,Z).
child(X,Z,Y) :- child_fact(X,Z,Y).

descendant(X,Y) :- child(X,Y,Z).
descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```



Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

17 Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# Simple Examples
## Family Relationships



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

A solution to both problems is found in the program.

rel02.pl

```
child_fact(oscar, karen, frank).
child_fact(mary, karen, frank).
child_fact(eve, anne, oscar).
child_fact(henry, anne, oscar).
child_fact(isolde, anne, oscar).
child_fact(clyde, mary, oscarb).

child(X,Z,Y) :- child_fact(X,Y,Z).
child(X,Z,Y) :- child_fact(X,Z,Y).

descendant(X,Y) :- child(X,Y,Z).
descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

The PROLOG programmer must *pay attention to processing* and *avoid infinite loops*

The program is no longer as elegant and simple as the—logically correct—first variant

Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Exercise 1 ([Ertel 2025], Exercise 5.4, p. 90)

1. Show by testing out that the theorem prover E (in contrast to PROLOG), given the knowledge base as in `rel.pl`, answers the query `?- descendant(clyde, karen).` correctly. Why is that?
2. Compare the answers of PROLOG and E for the query `?- descendant (X, Y).`.

Logic Programming with PROLOG
Hoàng Anh Đức
Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Exercise 1 ([Ertel 2025], Exercise 5.4, p. 90)

1. Show by testing out that the theorem prover E (in contrast to PROLOG), given the knowledge base as in `rel.pl`, answers the query `?- descendant(clyde, karen).` correctly. Why is that?
2. Compare the answers of PROLOG and E for the query `?- descendant (X, Y)..`

Semantics of PROLOG programs

- **Declarative semantics:** logical interpretation of the Horn clauses
- **Procedural semantics:** processing of the PROLOG program

# Simple Examples
## Family Relationships

- Execution begins at the top left with the query.
- Each edge represents a possible SLD resolution step with a complementary unifiable literal.
- While the search tree becomes infinitely deep by the recursive rule, the PROLOG execution terminates because the facts occur before the rule in the input data.

$$\neg\, child(e, o, a)$$

$$(child(e, o, a) \lor \neg\, child(e, a, o))$$

$$\boxed{child(e, a, o)} \qquad (child(e, a, o) \lor \neg\, child(e, o, a))$$

$$(child(e, o, a) \lor \neg\, child(e, a, o))$$

$$child(e, a, o) \qquad \ldots$$

Figure: PROLOG search tree for the execution of the program `rel.pl` with the query `child(eve,oscar,anne)`. The constants have been abbreviated to save space.


Logic Programming with PROLOG
Hoàng Anh Đức
Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
19 Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Simple Examples
## Family Relationships

- The PROLOG execution does not terminate
- The branches lead from the head (= positive literal) of a (Horn) clause to the subgoals. Because all subgoals of a clause must be solved, these are *"and branches"*. All other branches are *"or branches"*, of which at least one must be unifiable with its parent nodes.
  - **Remind:** In SLD resolution, *the literals of the current clause* are called *subgoals* and *the literals of the negated query* are the *goals*.
- The two outlined facts represent the solution to the query.
- The PROLOG interpreter does not terminate here, however, because it works by using a depth-first search with backtracking and thus first chooses the infinitely deep path to the far left.

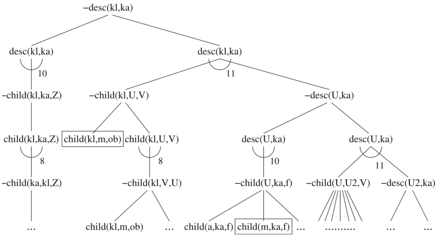



Figure: The and-or tree for the execution of the program `rel.pl` with the query `desc(clyde,karen)`


Logic Programming with PROLOG
Hoàng Anh Đức

Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
20 Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Execution Control and Procedural Elements



Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References



**Note**

As we have seen in the family relationship example, *it is important to control the execution of PROLOG*.

- *Avoiding unnecessary backtracking* especially can lead to large increases in efficiency. One means to this end is the *cut* operator. *By inserting an exclamation mark into a clause, we can prevent backtracking over this point.*
- Another possibility for execution control is the built-in predicate `fail`, which is never true.

# Execution Control and Procedural Elements

## Example 1 (Cut operator in PROLOG)

*max*$(X, Y,$ *Max*$)$ means “the maximum of two numbers $X$ and $Y$ is *Max*”

max.pl

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

maxwCut.pl

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

- Without cut.
- In query `?- max(2,3,Z), Z > 10.`, backtracking is employed because `Z = 3` and the second clause is tested for `max`, which is doomed to failure

- With cut.
- The second clause is only called if it is really necessary, that is, if the first clause fails.
- However, this optimization makes the program harder to understand.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
22 Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Example 2 (Predicate `fail` in PROLOG)

- In the family relationship example we can quite simply print out all children and their parents with the query

```
?- child_fact(X,Y,Z), write(X),
write(' is a child of '), write(Y),
write(' and '), write(Z), write('.'),
nl, fail.
```

- The corresponding output is

```
oscar is a child of karen and frank.
mary is a child of karen and frank.
eve is a child of anne and oscar.
henry is a child of anne and oscar.
isolde is a child of anne and oscar.
clyde is a child of mary and oscarb.
false.
```

where the predicate `nl` causes a line break in the output. What would be the output in the end without use of the `fail` predicate?

Logic Programming with PROLOG
Hoàng Anh Đức
Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
23 Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

## Example 3 (Negation as Failure)

- In the family relationship example, the query

```
?- child_fact(ulla,X,Y).
```

  would result `false.` because there are no facts about `ulla`.
- This answer is *not logically correct*. Specifically, *it is not possible to prove that there is no object with the name* `ulla`. Here the prover E would correctly answer "`No proof found.`"
- Thus if *PROLOG answers* `false.`, this only means that *the query $Q$ cannot be proved*. For this, however, $\neg Q$ *must not necessarily be proved*.

# Execution Control and Procedural Elements



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
25 Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

**Note**

Restriction to *Horn clauses* is *important for the procedural processing using SLD resolution*.

- Through the *singly determined positive literal per clause*, SLD resolution, and therefore the execution of PROLOG programs, have *a unique entry point into the clause*.
- This is *the only way it is possible* to *have reproducible execution of logic programs* and, therefore, *well-defined procedural semantics*.

## Example 4 (Statements that cannot be described by Horn clauses)

- Russell's paradox: *There is a barber who shaves everyone who does not shave himself*
- $Q \equiv \forall x\, \mathit{shaves}(\mathit{baber}, x) \Leftrightarrow \neg \mathit{shaves}(x, x) \equiv \forall x\, (\neg \mathit{shaves}(\mathit{baber}, x) \vee \neg \mathit{shaves}(x, x)) \wedge \forall x\, (\mathit{shaves}(x, x) \vee \mathit{shaves}(\mathit{baber}, x))$
- $Q$ contains the *non-Horn clause* $\mathit{shaves}(x, x) \vee \mathit{shaves}(\mathit{baber}, x)$

# Execution Control and Procedural Elements



Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

## Exercise 2 ([Ertel 2025], Exercise 5.1, p. 90)

Try to prove the theorem from Sect. 3.7 about the equality of left- and right-neutral elements of semi-groups with PROLOG. Which problems come up? What is the cause of this?

## Exercise 3 ([Ertel 2025], Exercise 5.5, p. 90)

Write as short a PROLOG program as possible that outputs 1024 ones.

# Lists

- A collection of *ordered data*.
- Has zero or more elements enclosed by square brackets and separated by commas (‘,’).

| **Example** | **Description** |
|---|---|
| `[A]` | A list with one element |
| `[]` | An empty list |
| `[34,tom,[2,3]]` | A list with three elements where the third element is a list of two elements |
| `[mia, love(honey), mia]` | A list with three elements where the first and last elements are identical |

- Like any object, a list can be unified with a variable

```
?- X = [Any, list, 'of elements'].
X = [Any, list, 'of elements'].
```

# Lists

- The construct `[Head|Tail]` separates the first element (`Head`) from the rest (`Tail`) of the list. (Thus, `Head` is always an atom and `Tail` is always a list.)

```
?- [H|T] = [A,2,2,B,3,4,5].
H = A,
T = [2, 2, B, 3, 4, 5].

?- [H|T] = [].
false.

?- X = [1|[2, 3]].
X = [1, 2, 3].

?- [1|2] = [1,2].
false.
```

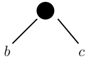
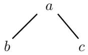
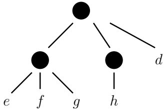
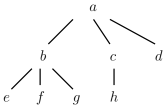
- By using *nested lists*, we can create *arbitrary tree structures*. Basically, in the trees where the inner nodes contain symbols, the symbol is the head of the list and the child nodes are the tail.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
28 Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Lists

## Example 5 (Tree Structures by Nested Lists)

| Tree | (Nested) Lists |
|---|---|
|  | `[b, c]` |
|  | `[a, b, c]` |
|  | `[[e, f, g], [h], d]` |
|  | `[a, [b, e, f, g], [c, h], d]` |

## Exercise 4 ([Ertel 2025], Exercise 5.7, p. 91)

Use function symbols instead of lists to represent the trees in Example 5.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
29 Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Lists



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Example 6 (List Processing)

*append*$(X, Y, Z)$ means “appending list $Y$ to the list $X$ and saving the result in $Z$”.

Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

30 Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

## Example 6 (List Processing)

*append*$(X, Y, Z)$ means "appending list $Y$ to the list $X$ and saving the result in $Z$".
The following program contains a *declarative (recursive) logical description* of the fact that `L3` results from appending `L2` to `L1`.

append.pl

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

- As long as `L` (= `[X|L1]`) is not empty, "appending `L2` to `L`" reduces to "appending `L2` to the tail `L1` of `L`" (and putting the head `X` of `L` as the first element of the result).

# Lists



## Example 6 (List Processing)

append.pl

```
1  append([],L,L).
2  append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

```
[trace] ?- append([1],[2],Z). % query
Call: (12) append([1], [2], _272) ? creep
Call: (13) append([], [2], _1612) ? creep
Exit: (13) append([], [2], [2]) ? creep
Exit: (12) append([1], [2], [1, 2]) ? creep
Z = [1, 2].
```

- Goal: `append([1], [2], _272).` (`Z = _272`: a variable whose value is not defined.)
- `[1]` is not empty ⇒ Unify with line 2: `[X|L1]/[1]`, `L2/[2]`, `[X|L3]/_272` (Now, `X = 1`, `L1 = []`, `L2 = [2]`, `L3 = _1612`)
- Subgoal: `append([], [2], _1612).`
- `[]` is empty ⇒ Unify with line 1: `[]/[]`, `L/[2]`, `L/_1612`. Thus, `_1612 = L = [2]`. Subgoal is solved.
- Therefore, `L3 = [2]`, which means `_272 = [X|L3] = [1, 2]`. Goal is solved.
- Output: `Z = [1, 2].`

# Lists



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
32 Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Example 7 (Naive Reverse)

*nrev*$(T, R)$ means “reversing the order of elements in the list $T$ and saving the result in $R$”.

## Example 7 (Naive Reverse)

*nrev*$(T, R)$ means "reversing the order of elements in the list $T$ and saving the result in $R$".
The following program describes how to recursively implement this predicate.

nrev.pl

```
nrev([],[]).
nrev([H|T],R) :- nrev(T,RT), append(RT,[H],R).
```

- As long as the list `L` (= `[H|T]`) is not empty, "reversing `L`" reduces to "reversing the tail `T` of `L`" (and appending the head `H` of `L` to the result).
- Indeed, this predicate is very inefficient due to calling `append`.




Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

32 Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
33 Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Example 7 (Naive Reverse)

nrev.pl

```
nrev([],[]).
nrev([H|T],R) :- nrev(T,RT), append(RT,[H],R).
```

```
[trace] ?- nrev([1,2], Z). % query
Call: (12) nrev([1, 2], _268) ? creep
Call: (13) nrev([2], _1594) ? creep
Call: (14) nrev([], _2406) ? creep
Exit: (14) nrev([], []) ? creep
Call: (14) append([], [2], _1594) ? creep
Exit: (14) append([], [2], [2]) ? creep
Exit: (13) nrev([2], [2]) ? creep
Call: (13) append([2], [1], _268) ? creep
Call: (14) append([], [1], _7296) ? creep
Exit: (14) append([], [1], [1]) ? creep
Exit: (13) append([2], [1], [2, 1]) ? creep
Exit: (12) nrev([1, 2], [2, 1]) ? creep
Z = [2, 1].
```

# Lists

Things go better when one proceeds using a temporary store, known as *the accumulator*, as follows:

| List | Accumulator |
| --- | --- |
| `[a,b,c,d]` | `[]` |
| `[b,c,d]` | `[a]` |
| `[c,d]` | `[b,a]` |
| `[d]` | `[c,b,a]` |
| `[]` | `[d,c,b,a]` |

*accrev*$(T, A, R)$ means "reversing the order of elements in the list $T$ (using $A$ as an "accumulator") and saving the result in $R$". The corresponding program is

accrev.pl

```
accrev([],A,A).
accrev([H|T],A,R) :- accrev(T,[H|A],R).
```

- As long as the list `L` (= `[H|T]`) is not empty, "reversing `L`" reduces to "reversing the tail `T` of `L`" (and putting the head `H` of `L` as the first element of the accumulator `A`).

# Lists

accrev.pl

```
accrev([],A,A).
accrev([H|T],A,R) :- accrev(T,[H|A],R).
```

```
[trace] ?- accrev([1,2], [], Z). % query
Call: (12) accrev([1, 2], [], _278) ? creep
Call: (13) accrev([2], [1], _278) ? creep
Call: (14) accrev([], [2, 1], _278) ? creep
Exit: (14) accrev([], [2, 1], [2, 1]) ? creep
Exit: (13) accrev([2], [1], [2, 1]) ? creep
Exit: (12) accrev([1, 2], [], [2, 1]) ? creep
Z = [2, 1].
```



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
35 Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Exercise 5 ([Ertel 2025], Exercise 5.6, p. 91)

Investigate the runtime behavior of the naive reverse predicate.

(a) Run PROLOG with the trace option and observe the recursive calls of `nrev`, `append`, and `accrev`.

(b) Compute the asymptotic time complexity of `append(L1,L2,L3)`, that is, the dependency of the running time on the length of the list for large lists. Assume that access to the head of an arbitrary list takes constant time.

(c) Compute the time complexity of `nrev(L,R)`.

(d) Compute the time complexity of `accrev(L,R)`.

(e) Experimentally determine the time complexity of the predicates `nrev`, `append`, and `accrev`, for example by carrying out time measurements (`time(+Goal)` gives inferences and CPU time.).

# Self-modifying Programs

- PROLOG programs are not fully compiled, rather, they are interpreted by the WAM. Therefore *it is possible to modify programs at runtime*. A program can even modify itself.
- With commands such as `assert` and `retract`, facts and rules can be added to the knowledge base or taken out of it.
- Assert predicates
  - `assert(X)`: Adds a new fact or clause to the database. Term is asserted as the last fact or clause with the same key predicate.
  - `asserta(X)`: Same as assert, but adds a clause at the beginning of the database.
  - `assertz(X)`: Exactly same as `assert(X)`.
- Retract predicates
  - `retract(X)`: Removes fact or clause `X` from the database.
  - `retractall(X)`: Removes all facts or clauses from the database for which the head unifies with `X`.


Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
37 Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Self-modifying Programs

A simple application of `asserta` is the addition of derived facts to the beginning of the knowledge base with the goal of avoiding a repeated, potentially time-expensive derivation.

## Example 8 (Family Relationship)

dynamic_rel.pl

```prolog
child_fact(oscar, karen, frank).
child_fact(mary, karen, frank).
child_fact(eve, anne, oscar).
child_fact(henry, anne, oscar).
child_fact(isolde, anne, oscar).
child_fact(clyde, mary, oscarb).

child(X,Z,Y) :- child_fact(X,Y,Z).
child(X,Z,Y) :- child_fact(X,Z,Y).

:- dynamic descendant/2.
descendant(X,Y) :- child(X,Y,Z), asserta(descendant(X,Y)).
descendant(X,Y) :- child(X,U,V), descendant(U,Y),
                   asserta(descendant(X,Y)).
```

# Self-modifying Programs

```
?- [dynamic_rel].
true.

?- descendant(clyde, karen).
true .

?- listing(descendant).
:- dynamic descendant/2.

descendant(clyde, karen).
descendant(mary, karen).
descendant(X, Y) :-
child(X, Y, Z),
asserta(descendant(X, Y)).
descendant(X, Y) :-
child(X, U, V),
descendant(U, Y),
asserta(descendant(X, Y)).

true.
```




Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

39 Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# Self-modifying Programs

- By manipulating rules with `assert` and `retract`, even programs that change themselves completely can be written. This idea became known under the term *genetic programming*. It allows the construction of arbitrarily flexible learning programs.
- In practice, however, it turns out that, due to the *huge number of senseless possible changes*, *changing the code by trial and error rarely leads to a performance increase*.
- *Systematic changing of rules*, on the other hand, *makes programming so much more complex* that, so far, such programs that extensively modify their own code have not been successful.
- *Machine learning* has been quite successful. However, only *very limited modifications* of the program code are being conducted here.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Self-modifying Programs

## Exercise 6 ([Ertel 2025], Exercise 5.8, p. 91)

The Fibonacci sequence is defined recursively by *fib*$(0) = 1$, *fib*$(1) = 1$ and *fib*$(n) =$ *fib*$(n-1) +$ *fib*$(n-2)$.

(a) Define a recursive PROLOG predicate `fib(N,R)` which calculates *fib*$(N)$ and returns it in `R`.

(b) Determine the runtime complexity of the predicate `fib` theoretically and by measurement.

(c) Change your program by using `asserta` such that unnecessary inferences are no longer carried out.

(d) Determine the runtime complexity of the modified predicate theoretically and by measurement (notice that this depends on whether `fib` was previously called).

(e) Why is `fib` with `asserta` also faster when it is started for the first time right after PROLOG is started?



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# A Planning Example

A farmer wants to bring a cabbage, a goat, and a wolf across a river, but his boat is so small that he can only take them across one at a time. The farmer thought it over and then said to himself: “If I first bring the wolf to the other side, then the goat will eat the cabbage. If I transport the cabbage first, then the goat will be eaten by the wolf. What should I do?”



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
42 A Planning Example
Constraint Logic Programming
References

# A Planning Example

plan.pl

```
start :- action(state(left,left,left,left),
                state(right,right,right,right)).

action(Start,Goal) :-
   plan(Start,Goal,[Start],Path),
   nl,write('Solution:'),nl,
   write_path(Path).
%   write_path(Path), fail. % all solutions output

plan(Start,Goal,Visited,Path) :-
   go(Start,Next),
   safe(Next),
   \+ member(Next,Visited),    % not(member(...))
   plan(Next,Goal,[Next|Visited],Path).
plan(Goal,Goal,Path,Path).

go(state(X,X,Z,K),state(Y,Y,Z,K)) :- across(X,Y). % farmer, wolf
go(state(X,W,X,K),state(Y,W,Y,K)) :- across(X,Y). % farmer, goat
go(state(X,W,Z,X),state(Y,W,Z,Y)) :- across(X,Y). % farmer, cabbage
go(state(X,W,Z,K),state(Y,W,Z,K)) :- across(X,Y). % farmer

across(left,right).
across(right,left).

safe(state(B,W,Z,K)) :- across(W,Z), across(Z,K).
safe(state(B,B,B,K)).
safe(state(B,W,B,B)).
```

# A Planning Example

- `state(Farmer,Wolf,Goat,Cabbage)` describes the current state of the world. Each variable has two possible values `left` and `right`.
  - For example, `state(left, left, right, right)` means the farmer and the wolf is on the left-hand side of the river and the goat and the cabbage is on the right-hand side.
- `across(X,Y)` means going from position `X` to position `Y`. Lines 22–23 indicate that there are only two possibilities for the pair `(X, Y)`, which are `(left, right)` and `(right, left)`.
- `go(Start,Next)` describes going from the state `Start` to the state `Next` (lines 17–20) using the predicate `across`.
  - Lines 17–20 describes all four possibilities for the predicate `go` (the farmer either go alone or carry with him exactly one of the three: the wolf, the goat, the cabbage).
  - `go(state(X,X,Z,K),state(Y,Y,Z,K)) :- across(X,Y))` means that the farmer and the wolf go across the river from position `X` (`left/right`) to position `Y` (`right/left`).




Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# A Planning Example

- `safe(Next)` checks if the state `Next` is "safe".
  - `safe(state(B,W,Z,K)) :- across(W,Z), across(Z,K).` (Line 25) means that a general state `state(B,W,Z,K)` is "safe" if either the wolf and the goat are not on the same position (`across(W,Z)`) or the goat and the cabbage are not on the same position (`across(Z,K)`).
  - `safe(state(B,B,B,K)).` (Line 26) means that the state where the farmer, the wolf, and the goat are on the same position is "safe". Similarly for line 27.
- `plan(Start,Goal,Visited,Path)` describes how to reach the state `Goal` from the state `Start`. The result—a list of states—is stored in `Path`. The list of visited states is stored in `Visited`.
- `plan` is implemented recursively (lines 10–15):
  - to go from `Start` to `Goal`,
  - go to a successor state `Next` (created using the predicate `go`),
  - test the safety of `Next` with the predicate `safe`,
  - test whether `Next` is already visited with the built-in predicate `member`,
  - and if `Next` is safe and not visited, recursively go from `Next` to `Goal` and putting `Next` at the beginning of the list `Visited`.

  The base case is at line 15, when `Start = Goal` and `Visited = Path`.




Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

45 A Planning Example

Constraint Logic Programming

References

# A Planning Example

For the query `?- start.`, we get the answers:

```
Solution:
Farmer and goat from left to right
Farmer from right to left
Farmer and cabbage from left to right
Farmer and goat from right to left
Farmer and wolf from left to right
Farmer from right to left
Farmer and goat from left to right
true ;
```

```
Solution:
Farmer and goat from left to right
Farmer from right to left
Farmer and wolf from left to right
Farmer and goat from right to left
Farmer and cabbage from left to right
Farmer from right to left
Farmer and goat from left to right
true ;
```



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
46 A Planning Example
Constraint Logic Programming
References

# A Planning Example

- For better understanding, we describe the definition of `plan` in logic:

$$\forall z\, \mathit{plan}(z,z) \wedge$$
$$\wedge\, \forall s \forall z \forall n \left[ \mathit{go}(s,n) \wedge \mathit{safe}(n) \wedge \mathit{plan}(n,z) \Rightarrow \mathit{plan}(s,z) \right]$$

  - The base case is $\forall z\, \mathit{plan}(z,z)$.
  - In the recursive call, for all starting state $s$, goal state $z$, and next state $n$, if we can go from $s$ to $n$ and $n$ is safe and we can recursively go from $n$ to $z$ then we can go from $s$ to $z$.
- This definition comes out significantly *more concise* than in PROLOG. There are two reasons for this:
  1. The output of the discovered plan is unimportant for logic.
  2. It is not really necessary to check whether the next state was already visited if unnecessary trips do not bother the farmer.
     - If, however, `\+ member(...)` is left out of the PROLOG program, then there is an infinite loop and PROLOG might not find a schedule even if there is one. The cause of this is *PROLOG's backward chaining search strategy*, which, according to the depth-first search principle, *always works on subgoals one at a time without restricting recursion depth*, and is therefore incomplete.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
47 A Planning Example
Constraint Logic Programming
References

# A Planning Example

- As in all planning tasks, *the state of the world changes as actions are carried out from one step to the next*.
- This suggests *sending the state as a variable to all predicates that depend on the state of the world*, such as in the predicate `safe`. The state transitions occur in the predicate `go`.
- This approach is called *situation calculus* [Russell and Norvig 2010].



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
48 A Planning Example
Constraint Logic Programming
References

# A Planning Example

## Exercise 7 ([Ertel 2025], Exercise 5.2, p. 90)

(a) Write a predicate `write_move(+State1, +State2)`, that outputs a sentence like “Farmer and wolf cross from left to right” for each boat crossing. `State1` and `State2` are terms of the form `state(Farmer, Wolf, Goat, Cabbage)`.

(b) Write a recursive predicate `write_path(+Path)`, which calls the predicate `write_move(+State1, +State2)` and outputs all of the farmer’s actions.

## Exercise 8 ([Ertel 2025], Exercise 5.3, p. 90)

(a) At first glance the variable `Path` in the predicate plan of the PROLOG program `plan.pl` is unnecessary because it is apparently not changed anywhere. What is it needed for?

(b) If we add a `fail` to the end of action in `plan.pl` (comment out Line 7 and un-comment Line 8), then all solutions will be given as output. Why is every solution now printed twice? How can you prevent this? (**Hint:** Look at the definition of the predicate `safe`.)



Additional Materials

PROLOG Systems and Implementations

Basic of PROLOG

Simple Examples

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

References

# Constraint Logic Programming

- The *programming of scheduling systems*, in which many (sometimes complex) logical and numerical conditions must be fulfilled, *can be very expensive and difficult with conventional programming languages*.
- This is precisely where *logic could be useful*.
- An approach is to simply *write all logical conditions in PL1 and then enter a query*. *Usually this approach fails miserably*. The reason is *the penguin problem* discussed in "Limitations of Logic". The fact `penguin(tweety)` does ensure that `penguin(tweety)` is true but does not rule out that `raven(tweety)` is also true. To rule this out with additional axioms is very inconvenient.
- *Constraint Logic Programming (CLP)* [Jaffar and Lassez 1987], which allows the *explicit formulation of constraints for variables*, offers an elegant and very efficient mechanism for solving this problem.
  - The *interpreter constantly monitors the execution of the program for adherence to all of its constraints*.
  - The programmer is fully relieved of the task of controlling the constraints, which in many cases can greatly simplify programming.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
50 Constraint Logic Programming
References

## Example 9 (Applying the CLP mechanism of GNU-PROLOG (The finite domain (FD) constraint solver))

The secretary of Albert Einstein High School has to come up with a plan for allocating rooms for final exams. He has the following information: the four teachers Mayer, Hoover, Miller and Smith give tests for the subjects German, English, Math, and Physics in the ascendingly numbered rooms 1, 2, 3 and 4. Every teacher gives a test for exactly one subject in exactly one room. Besides that, he knows the following about the teachers and their subjects.

(1) Mr. Mayer never tests in room 4.
(2) Mr. Miller always tests German.
(3) Mr. Smith and Mr. Miller do not give tests in neighboring rooms.
(4) Mrs. Hoover tests Mathematics.
(5) Physics is always tested in room number 4.
(6) German and English are not tested in room 1.

Who gives a test in which room?



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

# Constraint Logic Programming

raumplan.pl

```
%%% Run in GNU-PROLOG
start :-
    fd_domain([Mayer, Hoover, Miller, Smith],1,4),
    fd_all_different([Mayer, Miller, Hoover, Smith]),

    fd_domain([German, English, Math, Physics],1,4),
    fd_all_different([German, English, Math, Physics]),

    fd_labeling([Mayer, Hoover, Miller, Smith]),

    Mayer #\=4,                  % Mayer not in room 4
    Miller #= German,            % Miller tests German
    dist(Miller, Smith) #>= 2,   % Distance Miller/Smith >= 2
    Hoover #= Math,              % Hoover tests mathematics
    Physics #= 4,                % Physics in room 4
    German #\= 1,                % German not in room 1
    English #\= 1,               % English not in room 1
    nl,
    write([Mayer, Hoover, Miller, Smith]), nl,
    write([German, English, Math, Physics]), nl.
```

# Constraint Logic Programming

- GNU-PROLOG built-in predicates:
  - `fd_domain(Vars, Lower, Upper)` constraints each element `X` of `Vars` to take a value in `Lower..Upper`.
  - `fd_all_different(List)` constrains all variables in `List` to take distinct values.
  - `fd_labeling(Vars, Options)` assigns a value to each variable `X` of the list `Vars` according to the list of labeling options given by `Options`. This predicate is re-executable on backtracking. `fd_labeling(Vars)` is equivalent to `fd_labeling(Vars, [])`.
- The variables `Mayer,Hoover,Miller,Smith` as well as `German,English,Math,Physics` can each take on an integer value from 1 to 4 as the room number. (Lines 3–6.)
- A binding `Mayer = 1` and `German = 1` means that Mr. Mayer gives the German test in room 1.
- Lines 4 and 7 ensure that the four particular variables take on different values.
- Line 9 ensures that all variables are assigned a concrete value in the case of a solution. This line is not absolutely necessary here. If there were multiple solutions, however, only intervals would be output.
- In lines 11–17 the constraints are given, and the remaining lines output the room numbers for all teachers and all subjects in a simple format.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
53 Constraint Logic Programming
References

# Constraint Logic Programming

The program is loaded into GNU-PROLOG with `['raumplan.pl'].`, and with `start.` we obtain the output

```
[3,1,2,4]
[2,3,1,4]

true ?

yes
```

This output corresponds to the plan

| Room num. | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Teacher | Hoover | Miller | Mayer | Smith |
| Subject | Math | German | English | Physics |


Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
Constraint Logic Programming
References

## Exercise 9 ([Ertel 2025], Exercise 5.9, p. 91)

The following typical logic puzzle was supposedly written by Albert Einstein. Furthermore, he supposedly claimed that only 2% of the world's population is capable of solving it. The following statements are given.

- There are five houses, each painted a different color.
- Every house is occupied by a person with a different nationality.
- Every resident prefers a specific drink, smokes a specific brand of cigarette, and has a specific pet.
- None of the five people drinks the same thing, smokes the same thing, or has the same pet.
- Hints:
  - The Briton lives in the red house.
  - The Swede has a dog.
  - The Dane likes to drink tea.

# Constraint Logic Programming

- Hints (continue):
  - The green house is to the left of the white house.
  - The owner of the green house drinks coffee.
  - The person who smokes Pall Mall has a bird.
  - The man who lives in the middle house drinks milk.
  - The owner of the yellow house smokes Dunhill.
  - The Norwegian lives in the first house.
  - The Marlboro smoker lives next to the one who has a cat.
  - The man with the horse lives next to the one who smokes Dunhill.
  - The Winfield smoker likes to drink beer.
  - The Norwegian lives next to the blue house.
  - The German smokes Rothmanns.
  - The Marlboro smoker has a neighbor who drinks water.

Question: To whom does the fish belong?

(a) First solve the puzzle manually.

(b) Write a CLP program (for example with GNU-PROLOG) to solve the puzzle. Orient yourself with the room scheduling problem above.



Additional Materials
PROLOG Systems and Implementations
Basic of PROLOG
Simple Examples
Execution Control and Procedural Elements
Lists
Self-modifying Programs
A Planning Example
56 Constraint Logic Programming
References
57

# References


- Ertel, Wolfgang (2025). *Introduction to Artificial Intelligence*. 3rd. Springer. DOI: 10.1007/978-3-658-43102-0.
- Clocksin, William F. and Christopher S. Mellish (2013). *Programming in PROLOG Using the ISO Standard*. 3rd. Springer Science & Business Media. DOI: 10.1007/978-3-642-97005-4.
- Bratko, Ivan (2011). *Prolog programming for artificial intelligence*. 4th. Addison-Wesley.
- Russell, Stuart J. and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. 3rd. Pearson.
- Jaffar, Joxan and Jean-Louis Lassez (1987). "Constraint Logic Programming." In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 111–119. DOI: https://doi.org/10.1145/41625.41635.