

# VNU-HUS MAT1206E/3508: Introduction to AI

## Logic Programming with PROLOG In-class Discussion

Hoàng Anh Đức

Bộ môn Tin học, Khoa Toán-Cơ-Tin học  
Đại học KHTN, ĐHQG Hà Nội  
[hoanganhduc@hus.edu.vn](mailto:hoanganhduc@hus.edu.vn)



# Contents



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and Procedural Elements

Lists

Self-modifying Programs

A Planning Example

Constraint Logic Programming

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

# Additional Materials



Logic Programming  
with PROLOG

Hoàng Anh Đức

## 2 Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

Learn Prolog Now!

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

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

## 3 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- PROLOG = **P**rogramming in **L**ogic
- 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

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

4 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- PROLOG is a *declarative programming language*, i.e., the programmer declares *what* the program should accomplish without specifying *how* to achieve the result.
- PROLOG is based on *Horn clauses*.
- A PROLOG program consists of a *knowledge base* (*database*), which is simply a set of *facts* and *rules* about some problem domain.
  - A *knowledge base KB* of family relationships is coded as a PROLOG program

```
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, oscar).
```

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

5 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- The execution of a PROLOG program is initiated by a *query*, which is answered by proving that the query logically follows from the facts and rules in the program.

- Example query

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

- The query asks whether *eve* is a child of *anne* and *oscar*.
- The expected answer is *true* (because it is a fact in the knowledge base *KB*).
- How does PROLOG find the answer?
  - PROLOG tries to unify the query with the facts in the knowledge base *KB*.
  - There are six facts in the knowledge base.
  - Unification is attempted between the query and each of the complementary literals in the input data in order of occurrence. (In this example, the query unifies with the third fact.)
  - If one of the alternatives fails, this results in backtracking to the last branching point, and the next alternative is tested.

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

6

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Variables

- *Variables* begin with a Capital letter, or “\_”
  - For example, X, Tom, \_result
- “\_” is a nameless (anonymous) variable. We use it when we need to use a variable, but we're not interested in what PROLOG instantiates the variable to
- A variable can have a value

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

7 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## 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



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

8 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## 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.)



## 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.

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

10

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

Let's try some examples from "Learn Prolog Now!"  
(<https://www.let.rug.nl/bos/lpn/index.php>).

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

11 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Exercise 1

Given the following *KB*

```
1  woman(mia).  
2  woman(jody).  
3  woman(yolanda).  
4  playsAirGuitar(jody).  
5  party.
```

What is the expected answer to the following queries? Why?

```
?- woman(mia).
```

```
?- playsAirGuitar(mia).
```

```
?- playsAirGuitar(vincent).
```

```
?- tatooed(jody).
```

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

12 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Exercise 2

Given the following *KB*

```
1 happy(yolanda).  
2 listens2Music(mia).  
3 listens2Music(yolanda):- happy(yolanda).  
4 playsAirGuitar(mia):- listens2Music(mia).  
5 playsAirGuitar(yolanda):- listens2Music(yolanda).
```

What is the expected answer to the following queries? Why?

```
?- playsAirGuitar(mia).
```

```
?- playsAirGuitar(yolanda).
```

# Basic of PROLOG



## Exercise 3

Given the following *KB*

```
1 happy(vincent).  
2 listens2Music(butch).  
3 playsAirGuitar(vincent):- listens2Music(vincent),  
4     happy(vincent).  
5 playsAirGuitar(butch):- happy(butch).  
6 playsAirGuitar(butch):- listens2Music(butch).
```

What is the expected answer to the following queries? Why?

```
?- playsAirGuitar(vincent).
```

```
?- playsAirGuitar(butch).
```

Explain why we can replace the two rules in lines 6–7 by the single rule. (Note: Look up the meaning of the semicolon ; in PROLOG.)

```
playsAirGuitar(butch):- happy(butch);  
    listens2Music(butch).
```

Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

13 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

14 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Exercise 4

Given the following *KB*

```
1  woman(mia).  
2  woman(jody).  
3  woman(yolanda).  
4  
5  loves(vincent,mia).  
6  loves(marsellus,mia).  
7  loves(pumpkin,honey_bunny).  
8  loves(honey_bunny,pumpkin).
```

What is the expected answer to the following queries? Why?

```
?- woman(X).
```

```
?- loves(marsellus,X), woman(X).
```

# Basic of PROLOG



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

15 Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Exercise 5

Given the following *KB*

```
1 loves(vincent,mia).
2 loves(marsellus,mia).
3 loves(pumpkin,honey_bunny).
4 loves(honey_bunny,pumpkin).
5
6 jealous(X,Y):- loves(X,Z), loves(Y,Z).
```

What is the expected answer to each of the following queries?  
Why?

```
?- jealous(marsellus,W).
```

# Recursive Definitions

## Introduction



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

16 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- In PROLOG, we can define predicates recursively.
- A recursive definition requires:
  - At least one *base case* (non-recursive)
  - At least one *recursive case*
- Let's look at a classic example from "Learn PROLOG Now!": Eating and Digestion

# Recursive Definitions

Example: Eating



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

17 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

Consider the following knowledge base:

```
1  is_digesting(X,Y)  :-  just_ate(X,Y) .
2  is_digesting(X,Y)  :-
3      just_ate(X,Z),
4      is_digesting(Z,Y) .
5
6  just_ate(mosquito,blood(john)) .
7  just_ate(frog,mosquito) .
8  just_ate(stork,frog) .
```

- The definition of `is_digesting/2` is recursive
- It appears in both head and body of the second rule
- The first rule (base case) provides an “escape” from circularity

# Recursive Definitions

## Declarative Meaning



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

18 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- **Declarative meaning:** The logical meaning of the PROLOG knowledge base
- Base clause (non-recursive):
  - “If x has just eaten y, then x is now digesting y”
- Recursive clause:
  - “If x has just eaten z and z is digesting y, then x is digesting y too”
- This captures the intuition of indirect digestion through food chains

# Recursive Definitions

## Procedural Meaning



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

19 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- **Procedural meaning**: How PROLOG actually executes the queries
- For a query `is_digesting(X,Y)`, PROLOG:
  - First tries the base rule: “Has x just eaten Y?”
  - If that fails, tries the recursive rule by finding some Z where:
    - X has just eaten Z, AND
    - Z is digesting Y (recursive subgoal)

# Recursive Definitions

## Example Execution



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

20 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

For the query:

```
?- is_digesting(stork,mosquito).
```

PROLOG's execution:

1. Try base rule with  $X=\text{stork}$ ,  $Y=\text{mosquito}$ :  
 $\text{just\_ate}(\text{stork},\text{mosquito}) \Rightarrow \text{fails}$
2. Try recursive rule:
  - Find  $Z$  where  $\text{just\_ate}(\text{stork},Z) \Rightarrow Z = \text{frog}$
  - New subgoal:  $\text{is\_digesting}(\text{frog},\text{mosquito})$
  - Try base rule:  $\text{just\_ate}(\text{frog},\text{mosquito}) \Rightarrow \text{succeeds!}$
3. Query succeeds: Yes, stork is digesting mosquito

# Recursive Definitions

## The Importance of Base Cases



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

21 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Warning

Always include a base case in recursive definitions!

Consider this dangerous rule:

```
p :- p.
```

- Declaratively: “If property  $p$  holds, then property  $p$  holds” (logical)
- Procedurally: Creates an *infinite loop*
  - To prove  $p$ , I need to prove  $p$
  - To prove  $p$ , I need to prove  $p$
  - ...and so on, forever
- Without a base case, PROLOG won't terminate

# Recursive Definitions

## Another Example: Family Relationships



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

22 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

Consider the following knowledge base:

```
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).
```

The following query is not answered:

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

- The clause in line 8, which specifies symmetry of the child predicate, calls itself recursively without the possibility of termination.

# Recursive Definitions

## Another Example: Family Relationships



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

23 Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

This problem can be solved with the following new program.

rel01.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 descendant(X,Y) :- child(X,Y,Z).
9 descendant(X,Y) :- child(X,Z,Y).
10 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.**

# Recursive Definitions

## Another Example: Family Relationships



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

24 Recursive Definitions

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

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

# Recursive Definitions

## Another Example: Family Relationships



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

24 Recursive Definitions

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

```
1 child_fact(oscar, karen, frank).
2 child_fact(mary, karen, frank).
3 child_fact(eve, anne, oscar).
4 child_fact(henry, anne, oscar).
5 child_fact(isolde, anne, oscar).
6 child_fact(clyde, mary, oscarb).
7
8 child(X,Z,Y) :- child_fact(X,Y,Z).
9 child(X,Z,Y) :- child_fact(X,Z,Y).
10
11 descendant(X,Y) :- child(X,Y,Z).
12 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

# Execution Control and Procedural Elements



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

25 Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Note

As we have seen in the previous examples, *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



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

26 Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Example 1 (Cut operator in PROLOG)

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

max.pl

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

maxwCut.pl

```
1 max(X,Y,X) :- X >= Y, !.  
2 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.

# Execution Control and Procedural Elements



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

27 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?**

# Execution Control and Procedural Elements



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

28 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*.

# Lists



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

29 Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- 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



- A list can be decomposed into its *head* (first element) and *tail* (remaining elements) using the vertical bar operator ('|').
- For example, the list [A, B, C] can be decomposed as follows:

```
?- [Head|Tail] = [A, B, C].  
Head = A,  
Tail = [B, C].
```

- What are the head and the tail of the list [dead(z)]?
- **Note:** The empty list has neither a head nor a tail. That is, the empty list has no internal structure; for PROLOG, [] is a special, particularly simple, list.
  - What is the output of the following query?

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

# Lists



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

31 Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

## Exercise 6

Explain the purpose of each query and give PROLOG's expected answer. For each query below, state (1) what the query asks, (2) the expected result (true/false or variable bindings), and (3) a brief justification.

```
?- [X|Y] = [], dead(z), [2, [b, c]], [], Z].
```

```
?- [X,Y | W] =  
    [], dead(z), [2, [b, c]], [], Z].
```

```
?- [X1,X2,X3,X4 | Tail] =  
    [], dead(z), [2, [b, c]], [], Z].
```

```
?- [_,X,_,Y|_] =  
    [], dead(z), [2, [b, c]], [], Z].
```

```
?- [_,_,[_|X]|_] =  
    [], dead(z), [2, [b, c]], [], Z].
```

# Lists

## First recursive list program



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

32 Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

It's time to look at an example (from “Learn Prolog Now!”) of a recursive PROLOG program for lists: the predicate `member/2`.

- Goal: given an object `X` and a list `L`, decide whether `X` belongs to `L`.
- The standard definition (one fact and one recursive rule):

```
1 member(X, [X|T]).  
2 member(X, [H|T]) :- member(X, T).
```

- First clause (fact): “`X` is a member of a list if `X` is the head of that list.” (uses the `|` operator)
- Second clause (recursive rule): “`X` is a member of a list if `X` is a member of the tail of the list.”
- Declaratively this is straightforward: the two clauses capture membership directly from the structure of lists.

# Lists

## Procedural behaviour — examples



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

33 Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

Consider how PROLOG answers queries.

### ■ Immediate success:

```
?- member(yolanda,[yolanda,trudy,vincent,jules])
```

PROLOG succeeds immediately by unifying with the first clause.

### ■ Requires recursion/backtracking:

```
?- member(vincent,[yolanda,trudy,vincent,jules])
```

PROLOG tries the first clause (fails), uses the recursive clause repeatedly until the subgoal

```
member(vincent,[vincent,jules])
```

unifies with the first clause and succeeds.

# Lists

## Failure and termination



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

34 Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- If the queried element is not in the list, recursion eventually reaches the empty list and cannot proceed:

```
?- member(zed,[yolanda,trudy,vincent,jules]).
```

- PROLOG will derive successive goals

```
member(zed,[trudy,vincent,jules])  
member(zed,[vincent,jules])  
member(zed,[jules])  
member(zed,[])
```

and at `member(zed, [])`, neither clause applies (empty list cannot be split), so search stops and the answer is no.

# Lists

## Enumerating members & a small improvement



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

35 Lists

Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- `member/2` can be used with variables to enumerate elements:

```
?- member(X,[yolanda,trudy,vincent,jules]).  
X = yolanda ;  
X = trudy ;  
X = vincent ;  
X = jules ;  
no
```

- Small stylistic improvement: use anonymous variables for irrelevant parts

```
1 member(X,[X|_]).  
2 member(X,[_|T]) :- member(X,T).
```

Semantically identical, but clearer because each clause names only what matters.

# Self-modifying Programs



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

36 Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- 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`.

# Self-modifying Programs



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

37 Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

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 4 (Family Relationship)

dynamic\_rel.pl

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

# Self-modifying Programs



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

38 Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

```
?- [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.
```

# Self-modifying Programs



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

39 Self-modifying  
Programs

A Planning Example

Constraint Logic  
Programming

References

- 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.

# A Planning Example



## Exercise 7

Understand how to solve this problem using PROLOG.

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?”



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

40 A Planning Example

Constraint Logic  
Programming

References

# Constraint Logic Programming



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

41 Constraint Logic  
Programming

References

- 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.

# Constraint Logic Programming



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

42 Constraint Logic  
Programming

References

## Example 5 (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?

# Constraint Logic Programming



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

43 Constraint Logic  
Programming

References

raumplan.pl

```
1  %%% Run in GNU-PROLOG
2  start :-
3      fd_domain([Mayer, Hoover, Miller, Smith],1,4),
4      fd_all_different([Mayer, Miller, Hoover, Smith]),
5
6      fd_domain([German, English, Math, Physics],1,4),
7      fd_all_different([German, English, Math, Physics]),
8
9      fd_labeling([Mayer, Hoover, Miller, Smith]),
10
11     Mayer #\=4,                % Mayer not in room 4
12     Miller #= German,          % Miller tests German
13     dist(Miller, Smith) #>= 2, % Distance Miller/Smith >= 2
14     Hoover #= Math,            % Hoover tests mathematics
15     Physics #= 4,              % Physics in room 4
16     German #\= 1,              % German not in room 1
17     English #\= 1,             % English not in room 1
18     nl,
19     write([Mayer, Hoover, Miller, Smith]), nl,
20     write([German, English, Math, Physics]), nl.
```

# Constraint Logic Programming



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

44 Constraint Logic  
Programming

References

## ■ 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.

# Constraint Logic Programming



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs

A Planning Example

45 Constraint Logic  
Programming

References

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

# References



Logic Programming  
with PROLOG

Hoàng Anh Đức

Additional Materials

Basic of PROLOG

Recursive Definitions

Execution Control and  
Procedural Elements

Lists

Self-modifying  
Programs


A Planning Example

Constraint Logic  
Programming

46 References

 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.

 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>.