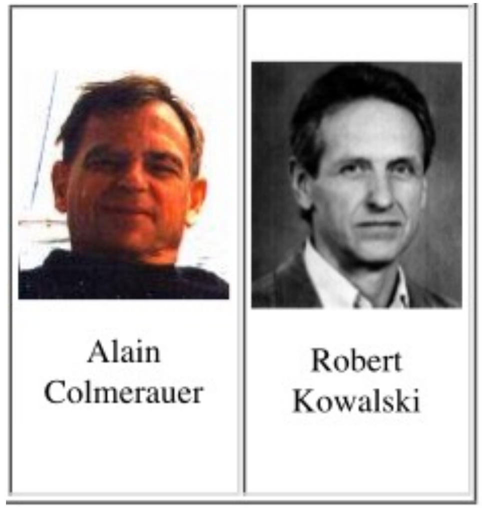
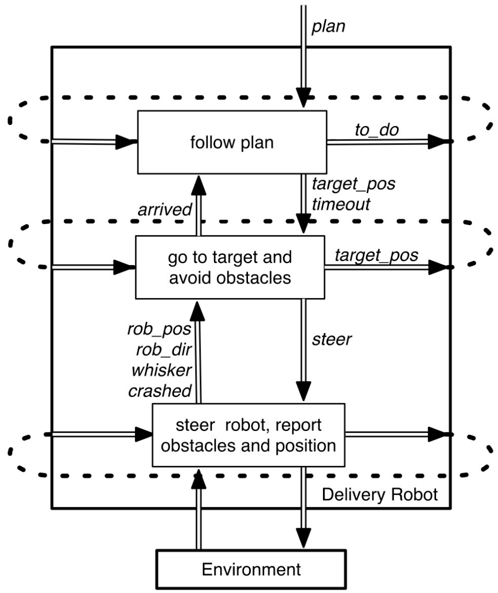
Introduction to Prolog

What is Prolog?



- Prolog = Programmation en Logique (Programming in Logic).
- Invented early seventies by Alain Colmerauer in France and Robert Kowalski in the UK

Why Prolog?



- Specifically designed for Artificial Intelligence
- The Prolog interpreter is based on a *theorem prover*
  - i.e. the programming language incorporates some reasoning
- Most useful in the upper layers of an agent architecture

Prolog

- Prolog is ***declarative*** programming language.
- You tell it what you want and it figures how to do it (sort of)
- The Prolog interpreter is an AI program

Relations

- Prolog programs specify relationships among objects.
- When we say, "John owns the book", we are declaring the ownership relation between two objects: John and the book.
- When we ask, "Does John own the book?", we are querying the relationship

Rules

- Relationships can also be rules such as

  Two people are sisters if

    both are female and

    they have the same parents

- This is a rule which allows us to find out about a relationship even if the relationship isn't explicitly declared

Programming in Prolog

- Declare facts describing explicit relationships between objects.
- Define rules describing implicit relationships between objects.
- Ask questions about relationships between objects

How Prolog Works

- Pose a question as a hypothesis
- See if facts and rules support the hypothesis
- Uses deductive reasoning
- Like a sophisticated database query language (called a deductive database)

Example: Representing Regulations

The rules for entry into a professional computer science society are set out below:

An applicant to the society is acceptable if he or she has been nominated by two established members of the society and is eligible under the terms below:

- The applicant graduated with a university degree.
- The applicant has two years of professional experience.
- The applicant pays a joining fee of $200.

An established member is one who has been a member for at least two years.

Facts

```
experience(fred, 3).
fee_paid(fred).
graduated(fred, unsw).
university(unsw).
nominated_by(fred, jim).
nominated_by(fred, mary).
joined(jim, 2016).
joined(mary, 2018).
current_year(2023).
```

Rules

```
acceptable(Applicant) :-
    nominated(Applicant),
    eligible(Applicant).

nominated(Applicant) :-
    nominated_by(Applicant, Member1),
    nominated_by(Applicant, Member2),
    Member1 \= Member2,
    current_year(ThisYear),
    joined(Member1, Year1), ThisYear >= Year1 + 2,
    joined(Member2, Year2), ThisYear >= Year2 + 2.

eligible(Applicant) :-
    graduated(Applicant, University), university(University),
    experience(Applicant, Experience), Experience >= 2,
    fee_paid(Applicant).
```

```
experience(fred, 3).
fee_paid(fred).
graduated(fred, unsw).
university(unsw).
nominated_by(fred, jim).
nominated_by(fred, mary).
joined(jim, 2016).
joined(mary, 2018).
current_year(2023).
```

Facts

- A fact such as, "Claude lectures in course COMP3411", is written as:

  lectures(claude, 3411).

- The names of relationships are in lower case letters.
- The name of the relationship appears as the first term and the objects appears as arguments to a function.
- A full stop "." must end a fact.
- *lectures(claude, 3411)* is also called a predicate.

The Prolog Database

- A collection of facts about a hypothetical computer science department.
- Together, these facts for Prolog's *database*.

```
lectures(ashesh, 2521).

lectures(mike, 9417).
lectures(claude, 3411).

lectures(claude, 3431).


studies(fred, 2521).
studies(jack, 3411).
studies(jill, 3431 ).
studies(jill, 9417).
studies(henry, 3431).
studies(henry, 9417).

year(fred, 1).
year(jack, 1).
year(jill, 4).
year(henry, 4).
```

Prolog Systems

- GNU Prolog, YAP (Yet Another Prolog), SWI-Prolog
- We will use SWI-Prolog
  - Works in Linux, macOS, Windows
  - Download from https://www.swi-prolog.org
  - Also in package library of most Linux distributions, MacPorts & HomeBrew
  - Interactive web version: https://swish.swi-prolog.org

Questions

Suppose we want to know if John lectures in course COMP1021.

- First load database file:

  ?- *[courses]* . *there is a file* courses.pl *in the current directory*

  true. *output from Prolog*

- We can ask:

  ?- *lectures(mike, 9417)*.

  true. *output from Prolog*

- To answer this question, Prolog consults its database to see if this is a known fact.
- Suppose we ask:

  ?- *lectures(fred, 9417)*.

  false. *output from Prolog*

- Prolog can't find a fact matching the question, so answer "false" is printed
- This query is said to have *failed*.

Variables

- Suppose we want to ask, "What subject does Ashesh teach?"
- This could be written as:
- Is there a subject, *X*, that Ashesh teaches?
- The variable, *X*, stands for an object that the questioner does not yet know about.
- To answer the question, Prolog has to find out the value of *X*, if it exists.
- As long as we do not know the value of a variable, it is said to be *unbound*.
- When a value is found, the variable is *bound* to that value.

Variables

- A variable must begin with a capital letter or "_".
- To ask Prolog to find the subject that Ashesh teaches, type:

  ?- *lectures(ashesh, Subject)*.

  Subject = 2521

- To ask which subjects that Claude teaches, ask:

  ?- *lectures(claude, Subject)*.

  Subject = 3411 **;**

  Subject = 3431.

- ***Prolog presents first answer and waits for user input.***
- ***Type ';' to get next answer.***
- ***Prolog can find all answers that satisfy a query***

Conjunctions of Goals

- How do we ask, "Does Ashesh teach Fred"?
- This can be answered by finding out if John lectures in a subject which Fred studies.

  lectures(ashesh, Subject), studies(fred, Subject).

- I.e. Ashesh lectures in subject, *Subject*, and Fred studies subject, *Subject*.
- *Subject* is a variable.
- The question consist of two *goals*.
- To find the answer, Prolog must find a single value for *Subject* that satisfies both goals.

Conjunctions

Who does Claude teach:

```
?- lectures(claude, Subject), studies(Student, Subject).
Subject = 3411
Student = jack ;
Subject = 3431
Student = Jill ;
Subject = 3431
Student = henry.
```

Conjunctions

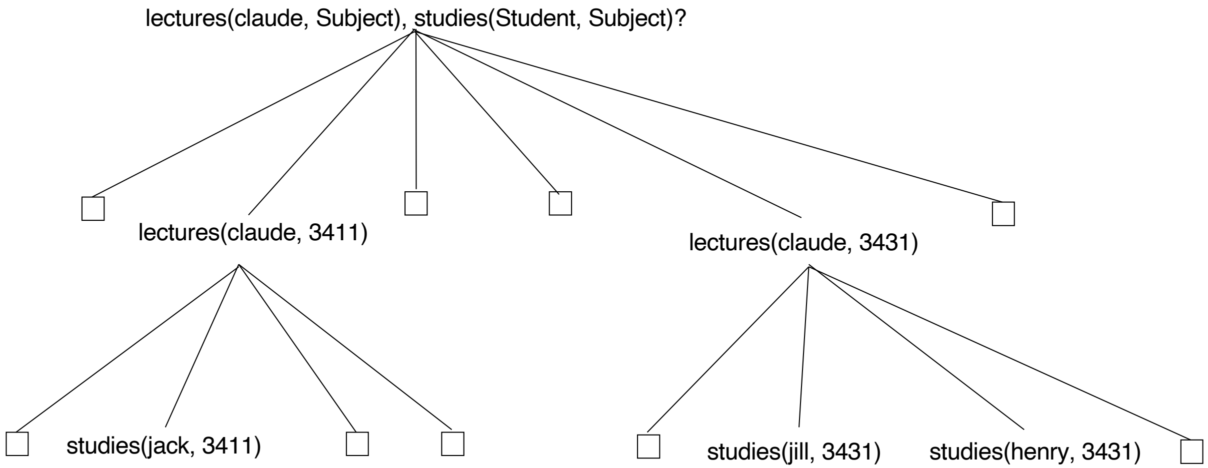
- Prolog solves problem by proceeding left to right and then backtracking.
- Given the initial query, Prolog tries to solve

```
lectures(claude, Subject)
```

- There are four lectures clauses, but only two have claude as first argument.
- Prolog chooses the first clause containing a reference to claude, i.e. `lectures(claude, 3411).`

Search Tree

- With *Subject = 3431*, tries to satisfy the next goal, *studies(Student, 3431)*.
- After solution found, Prolog backtracks and looks for alternative solutions.
- May go down branch containing *lectures(claude, 3431)* and then try *studies(Student, 3431)*.



Rules

- The previous question can be restated as a general rule:

  One person, *Teacher* teaches another person, *Student* if

  Teacher lectures subject, *Subject* and

  Student studies *Subject*.

- In Prolog this is written as:

```
teaches(Teacher, Student) :-
  lectures(Teacher, Subject),
  studies(Student, Subject).
?- teaches(ashesh, Student).
```

This is a clause

- Facts are *unit clauses* and rules are *non-unit clauses*.

Clause Syntax

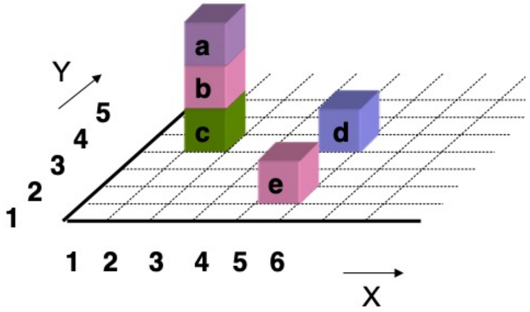
- ":–" means "if" or "is implied by". Also called "neck".
- The left hand side of the neck is the *head*.
- The right hand side of the neck is called the *body*.
- The comma, ",", separating the goals stands for *and*.

```
more_advanced(Student1, Student2) :-
   year(Student1, Year1),
   year(Student2, Year2),
   Year1 > Year2.
```

- Note the use of the *predefined predicate* ">".

```
more_advanced(henry, jack).
more_advanced(henry, X).
```

Declarative Meaning of a Program




```
Given facts and rules:

on(a, b).
on(b,c).
on(c, table).
…
above(B1, B2) :-
 on(B1, B2).
above(B1, B2) :-
 on(B1, B),
 above(B, B2).
```

```
What can we derive?

on(a, b).
on(b,c).
on(a, table).
…
above(a, b).
above(b, c).
above(a, c).
above(a, table)
…

All this constitutes the
declarative meaning or model
```

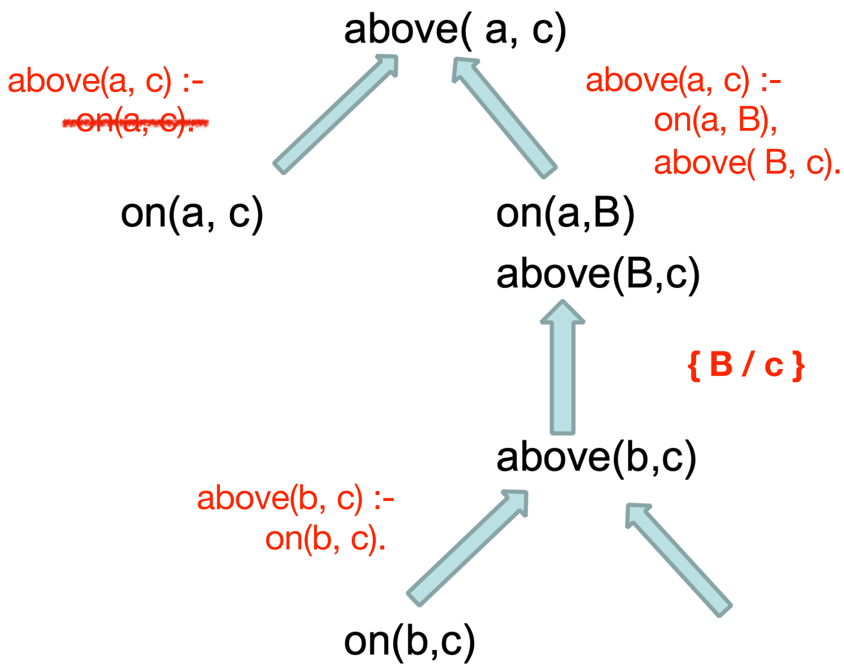
Proof Tree

```
on(a, b).
on(b, c).

above(B1, B2) :-
      on(B1, B2).

above(B1, B2) :-
      on(B1, B),
      above( B, B2).

?- above( a, c).
```



Structures

- Functional terms can be used to construct complex data structures.
- E.g. to say that John owns the book *"I, Robot"*, this may be expressed as:

```
owns(john, "I, Robot ").
```

- Often objects have a number of attributes.
- A book may have a title and an author.

owns(john, book("I, Robot ", asimov)).

- To be more accurate we should give the author's family ad given names.

```
owns(john, book("I, Robot ", author(asimov, isaac))).
```

Asking questions with structures

How do we ask,
"What books does John own that were written by someone called ″Asimov"?

```
?- owns(john, book(Title, author(asimov, GivenName))).
Title = "I, Robot"
GivenName = isaac

?- owns(john, Book).
Book = book("I, Robot", author(asimov, isaac))

?- owns(john, book(Title, Author)).
Title = "I, Robot"
Author = author(asimov, isaac)
```

Databases

- A database of books in a library contains facts of the form:

```
book(CatNo, Title, author(Family, Given)).
member(MemNo, name(Family, Given), Address).

loan(CatNo, MemNo, Borrowed).
```

- A member of the library may borrow a book.
- A "loan" records:
  - the catalogue number of the book
  - the number of the member
  - the borrow date
  - the due date

Database Structures

- Dates are stored as structures:

```
date(Year, Month, Day).
```

- E.g. **`date(2023, 2, 19)`** represents 19 February 2023.
- Names and addresses are all stored as character strings.
- Which books has a member borrowed?

```
has_borrowed(MemFamily, Title, CatNo) :-
  member(MemNo, name(MemFamily, _), _),
  loan(CatNo, MemNo, _, _),
  book(CatNo, Title, _).
```

Overdue Books

```
later(date(Y, M, D1), date(Y, M, D2)) :-   D1 > D2.
later(date(Y, M1, _), date(Y, M2, _)) :-   M1 > M2.
later(date(Y1, _, _), date(Y2, _, _)) :-   Y1 > Y2.

?- later(date(2023, 3, 19), date(2023, 2, 19)).

overdue(Today, Title, CatNo, MemFamily) :-
   loan(CatNo, MemNo, Borrowed),
   due_date(Borrowed, DueDate),
   later(Today, DueDate),
   book(CatNo, Title, _),
   member(MemNo, name(MemFamily, _), _).
```

Due Date

- *is* accepts two arguments
- The right hand argument must be an *evaluable* arithmetic expression.
- The term is evaluated and unified with the left hand argument.
- It *is not* an assignment statement
- Variables *cannot* be reassigned values.
- Arguments of comparison operators can also be arithmetic expressions.

```
due_date(date(Y, M1, D), date(Y, M2, D)) :-
  M1 < 12,
  M2 is M1 + 1.
due_date(date(Y1, 12, D), date(Y2, 1, D)) :-
  Y2 is Y1 + 1.
```

Reference

- Ivan Bratko, *Programming in Prolog for Artificial Intelligence*, 4th Edition, Pearson, 2013.