Artificial Intelligence

Prolog
What is Prolog?

- Invented early 70s by Alain Colmerauer in France and Robert Kowalski in Britain.
- Programmation en Logique (Programming in Logic).
- differs from most common programming languages
- is a declarative language
  - programmer specifies a goal to be achieved
  - Prolog system works out how to achieve it
What is Prolog?

• traditional programming languages are said to be **procedural**
• procedural programmer must specify in detail how to solve a problem:
  
  - mix ingredients;
  - beat until smooth;
  - bake for 20 minutes in a moderate oven;
  - remove tin from oven;
  - put on bench;
  - close oven;
  - turn off oven;

• in purely **declarative** languages, the programmer only states what the problem is and leaves the rest to the language system
Applications of Prolog

- intelligent data base retrieval
- natural language understanding
- expert systems
- specification language
- machine learning
- robot planning
- automated reasoning
- problem solving
- …
Relations

• Prolog programs specify *relationships* among objects and properties of objects.

• When we say, "John owns the book", we are declaring the ownership relationship between two objects: John and the book.

• When we ask, "Does John own the book?" we are trying to find out about a relationship.

• Relationships can also be specified as rules such as:
  
  Two people are sisters *if*
  
  *they are both female* and *they have the same parents*.

• A rule allows us to find out about a relationship even if the relationship isn't explicitly stated as a fact.
Programming in Prolog

• declare facts describing *explicit relationships* between objects and properties objects might have (e.g. Mary likes pizza, grass has _colour_ green)
• define rules defining *implicit relationships* between objects (e.g. the sister rule above) and/or rules defining implicit object properties (e.g. X is a parent if there is a Y such that Y is a child of X).

One then uses the system by:
• asking questions about relationships between objects, and/or about object properties (e.g. does Mary like pizza? is Joe a parent?)
• Properties of objects, or relationships between objects;
• "Dr Turing lectures in course 9020", is written in Prolog as: 
  lectures(turing, 9020).

• Notice:
  – names of properties/relationships begin with lower case letters.
  – the relationship name appears as the first term
  – objects appear as comma-separated arguments within parentheses.
  – A period "." must end a fact.
  – objects also begin with lower case letters. They also can begin with
digits (like 9020), and can be strings of characters enclosed in
quotes (as in reads(fred, "War and Peace")).

• lectures(turing, 9020). is also called a predicate
Facts

• Facts about a hypothetical computer science department:

  % lectures(X, Y): person X lectures in course Y
  lectures(turing, 9020).
  lectures(codd, 9311).
  lectures(backus, 9021).
  lectures(ritchie, 9201).
  lectures(minsky, 9414).
  lectures(codd, 9314).

  % studies(X, Y): person X studies in course Y
  studies(fred, 9020).
  studies(jack, 9311).
  studies(jill, 9314).
  studies(jill, 9414).
  studies(henry, 9414).
  studies(henry, 9314).

  %year(X, Y): person X is in year Y
  year(fred, 1).
  year(jack, 2).
  year(jill, 2).
  year(henry, 4).
Queries

- Once we have a database of facts (and, soon, rules) we can ask questions about the stored information.
- Suppose we want to know if Turing lectures in course 9020. We can ask:

  | % prolog -s facts03.pro |
  | (multi-line welcome message) |
  | ?- lectures(turing, 9020). |
  | Yes/True |
  | ?- <control-D> |
  | % |

  facts03 loaded into Prolog
  "?-" is Prolog's prompt output from Prolog
  hold down control & press D to leave Prolog

- Notice:
  - In SWI Prolog, queries are terminated by a full stop.
  - To answer this query, Prolog consults its database to see if this is a known fact.
  - In example dialogues with Prolog, the text in green italics is what the user types.
Query

?- lectures(codd, 9020).
No/fail

• if answer is Yes/true, the query succeeded
• if answer is No/fail, the query failed
• The use of lower case for codd is critical.
• Prolog is not being intelligent about this - it would not see a difference between this query and lectures(fred, 9020). or lectures(xyzzy, 9020). though a person inspecting the database can see that fred is a student, not a lecturer, and that xyzzy is neither student nor lecturer.
Variables

- Question: "What course does Turing teach"?
- This could be written as:
  Is there a course, X, that Turing teaches?
- The variable X stands for an object which the questioner does not know about yet.
- 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 said to *bound* to that value.
- The name of a variable must begin with a capital letter or an underscore character, "_".
Variables

• To ask Prolog to find the course which Turing teaches, the following query is entered:

```prolog
?- lectures(turing, Course).
```

Course = 9020 ← output from Prolog

• To ask which course(s) Prof. Codd teaches, we may ask,

```prolog
?- lectures(codd, Course).
```

Course = 9311 ; ← type ";" to get next solution
Course = 9314 ;

No

• Prolog can find all possible ways to answer a query, unless you explicitly tell it not to (see cut, later).
Conjunctions of Goals

• How do we ask, "Does Turing teach Fred"?
• This means finding out if Turing lectures in a course that Fred studies.

?- lectures(turing, Course), studies(fred, Course).
• To answer this question, Prolog must find a single value for Course, that satisfies both goals.
• Read the comma, ",", as **and**.
• However, note that Prolog will evaluate the two goals left-to-right. This is sometimes referred to as "conditional-and".
Backtracking

- Who does Codd teach?

`?- lectures(codd, Course), studies(Student, Course).`

Course = 9311 Student = jack ;
Course = 9314 Student = jill ;
Course = 9314 Student = henry ;

- Processes left to right and then *backtracking*.
- Prolog starts by trying to solve `lectures(codd, Course)`
- six lectures clauses, only two have codd as their first argument.
- Uses the first clause that refers to codd: `lectures(codd, 9311)`.
- It tries the next goal, `studies(Student, 9311)`.
- It finds the fact `studies(jack, 9311).` and hence the first solution: `(Course = 9311, Student = jack)`
Backtracking

• After the first solution is found, Prolog retraces its steps up the tree and looks for alternative solutions.
• First it looks for other students studying 9311 (but finds none).
• Then it
  – backs up
  – rebinds Course to 9314,
  – goes down the lectures(codd, 9314) branch
  – tries studies(Student, 9314),
  – finds the other two solutions: (Course = 9314, Student = jill) and (Course = 9314, Student = henry).
Backtracking

Proof tree:

```
lectures(codd, Course), studies(Student, Course)

lectures(codd, 9311)

studies(jack, 9311)

lectures(codd, 9314)

studies(jill, 9314)

studies(henry, 9314)
```
Rules

- The previous question can be restated as a general rule: *One person, Teacher, teaches another person, Student if Teacher lectures in a course, Course and Student studies Course.*

- In Prolog this is written as:
  
  ```prolog
  teaches(Teacher, Student) :-
    lectures(Teacher, Course),
    studies(Student, Course).
  ?- teaches(codd, Student).
  ```

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

• "\:-" means "if" or "is implied by". Also called the neck symbol.
• The left hand side of the neck is called the head.
• The right hand side of the neck is called the body.
• The comma, ",," separating the goals is stands for and.
• Another rule, using one of the predefined predicate ">".

   more_advanced(S1, S2) :-
   year(S1, Year1),
   year(S2, Year2),
   Year1 > Year2.
Tracing Execution

?- trace.
Yes
[trace] ?- more_advanced(henry, fred).
Call: more_advanced(henry, fred) ?
Call: year(henry, _L205) ?
Exit: year(henry, 4) ?
Call: year(fred, _L206) ?
Exit: year(fred, 1) ?
^ Call: 4>1 ?
^ Exit: 4>1 ?
Exit: more_advanced(henry, fred) ?
Yes
[debug] ?- notrace.

bind S1 to henry, S2 to fred
test 1st goal in body of rule succeeds, binds Year1 to 4
test 2nd goal in body of rule succeeds, binds Year2 to 1
test 3rd goal: Year1 > Year2 succeeds
Succeeds
More?

• Suppose we have the following facts and rule:
  bad_dog(fido).
  bad_dog(Dog) :-
      bites(Dog, Person),
      is_person(Person),
      is_dog(Dog).
  bites(fido, postman).
  is_person(postman).
  is_dog(fido).
There are two ways to prove bad_dog(fido):
(a) it’s there as a fact; and
(b) it can be proven using the bad_dog rule:
?- bad_dog(fido).

More? ;
Yes

More? means Yes and prompts us to type ; if we want to check for another proof. The Yes that follows means that a second proof was found. Alternatively, we can just press the "return" key if we are not interested in whether there is another proof.
Structures

• Functional terms can be used to construct complex data structures.
• If we want to say that John owns the novel Tehanu, we can write: owns(john, 'Tehanu').
• Objects have a number of attributes:
  owns(john, book('Tehanu', leguin)).
• The author LeGuin has attributes too:
  owns(john, book('Tehanu', author (leguin, ursula))).
• The arity of a term is the number of arguments it takes.
• all versions of owns have arity 2, but the detailed structure of the arguments changes.
• gives(john, book, mary). is a term with arity 3.
Asking Questions with Structures

• How do we ask, "What books does John own which were written by someone called LeGuin"?
  
  \[
  \text{?- owns(john, book(Title, author(leguin, GivenName))).}
  \]
  
  Title = 'Tehanu' GivenName = ursula

• What books does John own?
  
  \[
  \text{?- owns(john, Book).}
  \]
  
  Book = book('Tehanu', author(leguin, ursula))

• What books does John own?
  
  \[
  \text{?- owns(john, book(Title, Author)).}
  \]
  
  Title = 'Tehanu' Author = author(leguin, ursula)

• Prolog performs a complex matching operation between the structures in the query and those in the clause head.
Library Database Example

• A database of books in a library contains facts of the form
  
  \[
  \text{book}(\text{CatalogNo}, \text{Title}, \text{author}(\text{Family}, \text{Given})). \\
  \text{libmember}(\text{MemberNo}, \text{name}(\text{Family}, \text{Given}), \text{Address}). \\
  \text{loan}(\text{CatalogNo}, \text{MemberNo}, \text{BorrowDate}, \text{DueDate}).
  \]

• A member of the library may borrow a book.
• A "loan" records:
  – the catalogue number of the book
  – the number of the member
  – the date on which the book was borrowed
  – the due date
Library Database Example

- Dates are stored as structures:
  `date(Year, Month, Day)`
- e.g. `date(2008, 6, 16)` represents 16 June 2008.
- which books has a member borrowed?

```prolog
borrowed(MemFamily, Title, CatalogNo) :-
  libmember(MemberNo, name(MemFamily, _, _), _),
  loan(CatalogNo, MemberNo, _, _),
  book(CatalogNo, Title, _).
```

- The underscore or "don't care" variables (__) are used because for the purpose of this query we don't care about the values in some parts of these structures.
Comparing Two Terms

• we would like to know which books are overdue; how do we compare dates?

%later(Date1, Date2) if Date1 is after Date2:
later(date(Y, M, Day1), date(Y, M, Day2)) :-
    Day1 > Day2.
later(date(Y, Month1, _), date(Y, Month2, _)) :-
    Month1 > Month2.
later(date(Year1, _, _), date(Year2, _, _)) :-
    Year1 > Year2.

• This rule has three clauses: in any given case, only one clause is appropriate. They are tried in the given order.
• This is how disjunction (or) is often achieved in Prolog.
Overdue Books

% overdue(Today, Title, CatalogNo, MemFamily):
% given the date Today, produces the Title, CatalogNo,
% and MemFamily of all overdue books.

overdue(Today, Title, CatalogNo, MemFamily) :-
  loan(CatalogNo, MemberNo, _, DueDate),
  later(Today, DueDate),
  book(CatalogNo, Title, _),
  libmember(MemberNo, name(MemFamily, _), _).
Due Date

- Assume the loan period is one month, find the due date from today:

```prolog
%due_date(Today, DueDate).
due_date(date(Y, Month1, D), date(Y, Month2, D)) :-
    Month1 < 12,
    Month2 is Month1 + 1.
due_date(date(Year1, 12, D), date(Year2, 1, D)) :-
    Year2 is Year1 + 1.
```
The *is* operator

- The right hand argument of *is* must be an arithmetic expression that can be evaluated right now (no unbound variables).
- This expression is evaluated and bound to the left hand argument.
- *is* is not a C-style assignment statement:
  - X is X + 1 won't work!
  - except via backtracking, variables can only be bound once, using *is* or any other way
The *is* operator

- = does **not** cause evaluation of its arguments:

```?- X = 2, Y = X + 1.
X = 2
Y = 2+1
?- X = 2, Y is X + 1.
X = 2
Y = 3```

- Use *is* if and only if you need to evaluate something:

```
X is 1 BAD! - nothing to evaluate
X = 1 GOOD!
```
Binary Trees

- In the library database example, some complex terms contained other terms, for example, book contained name.
- The following term also contains another term, this time one similar to itself:
  \[
  \text{tree(tree(L1, jack, R1), fred, tree(L2, jill, R2))}
  \]
- The variables L1, L2, R1, and R2 should be bound to sub-trees.
Recursive Structures

- A term that contains another term that has the same principal functor (in this case tree) is said to be recursive.
- Biological trees have leaves. For us, a *leaf* is a node with two empty branches:
Another Tree Example

- \( \text{tree(tree(empty, 7, empty), ' + ', tree(tree(empty, 5, empty), '-' , tree(empty, 3, empty)))} \)
Recursive Programs for Recursive Structures

- A binary tree is either empty or contains some data and a left and right subtree which are also binary trees.

\[
\text{is\_tree}(\text{empty}). \quad \text{trivial branch}
\]
\[
\text{is\_tree}(\text{tree}(\text{Left}, \text{Data}, \text{Right})) :- \quad \text{recursive branch}
\]
\[
\text{is\_tree}(\text{Left}),
\text{some\_data}(\text{Data}),
\text{is\_tree}(\text{Right}).
\]

- A non-empty tree is represented by a 3-arity term.
- Any recursive predicate must have:
  - (at least) one \textbf{recursive branch/rule} (or it isn't recursive :-) ) and
  - (at least) one non-recursive or \textbf{trivial branch} (to stop the recursion going on for ever).
Recursive Programs for Recursive Structures

• Let us define (or measure) the size of tree (i.e. number of nodes):

\[
\text{tree\_size}(\text{empty}, 0).
\]
\[
\text{tree\_size}(\text{tree}(L, _, R), \text{Total\_Size}) :\text{-}
\]
\[
\text{tree\_size}(L, \text{Left\_Size}),
\]
\[
\text{tree\_size}(R, \text{Right\_Size}),
\]
\[
\text{Total\_Size is Left\_Size + Right\_Size + 1}.
\]
Lists

• A list may be nil (i.e. empty) or it may be a term which has a head and a tail.
• The head may be any term or atom.
• The tail is another list.
• We could define lists as follows:

  is_list(nil).
  is_list(list(Head, Tail)) :-
    is_list(Tail).

• A list of numbers [1, 2, 3] would look like: list(1, list(2, list(3, nil)))
• Since lists are used so often, Prolog has a special notation:

  [1, 2, 3] = .(1, .(2, .(3, [])))

?- X = .(1, .(2, .(3, []))).
X = [1, 2, 3]
List Constructor

• Within the square brackets [ ], the symbol | acts as an operator to construct a list from an item and another list.

?- \( X = [1 \mid [2, 3]] \).
\( X = [1, 2, 3] \).

?- \( \text{Head} = 1 \), \( \text{Tail} = [2, 3] \), \( \text{List} = [\text{Head} \mid \text{Tail}] \).
\( \text{List} = [1, 2, 3] \).
List Examples

?- [X, Y, Z] = [1, 2, 3].
X = 1  Y = 2  Z = 3

?- [X / Y] = [1, 2, 3].
X = 1  Y = [2, 3]

?- [X / Y] = [1].
X = 1  Y = []

?- [X, Y / Z] = [fred, jim, jill, mary].
X = fred  Y = jim  Z = [jill, mary]

?- [X / Y] = [[a, f(e)], [n, m, [2]]].
X = [a, f(e)]  Y = [[n, m, [2]]]
List Membership

- A term is a member of a list if
  - the term is the same as the head of the list, or
  - the term is a member of the tail of the list.
- In Prolog:
  member(X, [X | _]).
  member(X, [_ | Y]) :-
    member(X, Y).
- Member is actually predefined in Prolog.
• Suppose we want to take two lists, like [1, 3] and [5, 2] and concatenate them to make [1, 3, 5, 2]

concat([], L, L).
concat([Item | Tail1], L, [Item | Tail2]) :-
    concat(Tail1, L, Tail2).
An Application of Lists

• Find the total cost of a list of items:

% cost data:

\text{cost(cornflakes, 230)}.$
\text{cost(cocacola, 210)}.$
\text{cost(chocolate, 250)}.$
\text{cost(crisps, 190)}.$

?- \text{total\_cost([cornflakes, crisps], X)}.$
\text{X} = 420
An Application of Lists

total_cost([], 0).
total_cost([Item|Rest], Cost) :-
    cost(Item, ItemCost),
    total_cost(Rest, CostOfRest),
    Cost is ItemCost + CostOfRest.

• How about if we change the recursive branch:

total_cost([Item|Rest], Cost) :-
    total_cost(Rest, CostOfRest),
    cost(Item, ItemCost),
    Cost is ItemCost + CostOfRest.
Negation as Failure

• Build-in predicate not.

?- not(lectures(turing, 9020)).
Not/fail
Remove duplicates

?- remove_dups([1,2,3,1,3,4], X).
X = [2, 1, 3, 4]

% remove_dups(+List, -NewList):
remove_dups([], []).
remove_dups([First | Rest], NewRest) :-
    member(First, Rest),
    remove_dups(Rest, NewRest).
remove_dups([First | Rest], [First | NewRest]) :-
    not(member(First, Rest)),
    remove_dups(Rest, NewRest).
Controlling Execution
The Cut Operator

• Sometimes we need a way to prevent Prolog finding all solutions, i.e. a way to stop backtracking.
• The cut operator, written !, is a built-in goal that prevents backtracking.
• It turns Prolog from a nice declarative language into a hybrid monster.
The Cut Operator!

• Cut prunes the search tree, prevents backtracking:
  – Once the cut operator has been passed when evaluating a predicate, no new variable instantiations are allowed to those variables which are bound at that point in time.
  – Uninstantiated variables can still be instantiated after the cut operator has been processed.
  – Backtracking can still take place, but only for those uninstantiated variables.

• If the goal(s) to the right of the cut fail then the entire clause fails and the goal which caused this clause to be invoked fails.

• In particular, alternatives for Course are not explored.
The Cut operator

\[ \text{lectures}(\text{codd, Course}), !, \text{studies}(\text{Student, Course}) \]

- \text{lectures}(\text{codd, 9311})
  - \text{studies}(\text{jack, 9311})
  - \text{studies}(\text{other students})

- \text{lectures}(\text{codd, 9314})
  - \text{studies}(\text{jill, 9314})
  - \text{studies}(\text{henry, 9314})
Cut example

?- lectures(codd, X).
X = 9311 ;
X = 9314 ;
No
?- lectures(codd, X), ! .
X = 9311.
Cut example

• max, without cut:
  \%
  max(A, B, C) binds C to the larger of A and B.
  max(A, B, A) :-
    A > B.
  max(A, B, B) :-
    A =< B.
• max, with cut:
  max(A, B, A) :-
    A > B, !.
  max(A, B, B).
Cut example

- remove_dups, with cut:
  
  \[
  \text{remove\_dups}([], []).
  \]
  
  \[
  \text{remove\_dups}([\text{First} \mid \text{Rest}], \text{NewRest}) :-
  \]
  
  \[
  \text{member}(<\text{First}, \text{Rest}>,)
  \]
  
  .
  
  \[
  \text{remove\_dups}(\text{Rest}, \text{NewRest}).
  \]
  
  \[
  \text{remove\_dups}([\text{First} \mid \text{Rest}], [\text{First} \mid \text{NewRest}]) :-
  \]
  
  \[
  \text{remove\_dups}(\text{Rest}, \text{NewRest}).
  \]
Exercises

• Reversing Lists:
  – reverse(A, B): B is the reversed list of A.
• Viết chương trình duyệt theo chiều rộng xem có đường đi từ 1 đỉnh này đến 1 đỉnh khác (adjacency list representation adjacent(1, [2, 4, 5]).):
  – reachable(A, B): Có đường đi từ A đến B không?
  – path(A, B, Path): in ra đường đi từ A đến B.
• Missionaries and Cannibals