Introduction to Prolog

References:
1. Bratko chapters 1-5
2. iProlog Programmers Reference Manual

Aim:
To introduce enough of Prolog to allow students to do the assignment work in this course, thereby gaining some experience of AI programming.

Plan:
• What is Prolog? Programming with relations.
• Facts
• Queries
• Variables
• Conjunctions of goals, backtracking
• Rules
• Structures
• Recursive programs, trees
• Lists
• Controlling execution - the cut

What is Prolog?
• invented early seventies by Alain Colmerauer in France.
• Pro grammation en Logique (Programming in Logic).
• differs from the most common programming languages
• is a declarative language
  – programmer specifies a goal to be achieved
  – Prolog system works out how to achieve it
• 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

Some applications of Prolog:
• intelligent data base retrieval
• natural language understanding
• expert systems
• specification language
• machine learning
• robot planning
• automated reasoning
• problem solving

Introduction to Prolog

Reference: Bratko Chapter 1

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 querying the relationship.
• Relationships can also be 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

- declaring some facts about objects and their relationships
- defining rules about objects and their relationships
- asking questions about objects and their relationships

### Facts

- Properties of objects or relationships between objects
- "Dr Turing lectures in course 9020", is written as:
  \[
  \text{lectures(turing, 9020)}.
  \]

- **Notice:**
  
  - names of properties/relationships begin with lower case letters
  - the relationship name appears as the first term
  - objects appear as arguments in parentheses
  - a period "." must end a fact
  - objects must also begin with lower case letters

- \[
  \text{lectures(turing, 9020)}.
  \]
  is also called a **predicate**.

### Database

Facts about a hypothetical computer science department:

\[
\begin{align*}
& \text{% lectures(X, Y): X lectures course Y} \\
& \text{lectures(turing, 9020).} \\
& \text{lectures(codd, 9311).} \\
& \text{lectures(backus, 9201).} \\
& \text{lectures(ritchie, 9201).} \\
& \text{lectures(minsky, 9414).} \\
& \text{lectures(codd, 9314).} \\
& \text{% studies(X, Y): X studies course Y} \\
& \text{studies(fred, 9020).} \\
& \text{studies(jack, 9311).} \\
& \text{studies(jill, 9314).} \\
& \text{studies(jill, 9414).} \\
& \text{studies(henry, 9414).} \\
& \text{studies(henry, 9314).} \\
& \text{% year(X, Y): X is in year Y} \\
& \text{year(fred, 1).} \\
& \text{year(jack, 2).} \\
& \text{year(jill, 2).} \\
& \text{year(henry, 4).}
\end{align*}
\]

Together, when loaded into Prolog, these facts form Prolog's **database**.

These particular facts are available at

http://http.cse.unsw.edu.au/~billw/cs9414/notes/prolog/facts03

in case you would like experiment with them.
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.

```
lectures(turing, 9020)?
** yes
```

- Things to notice:
  - In iProlog, queries are terminated by a question mark, `?`.
  - Other Prolog implementations use different conventions.
  - To answer the question, Prolog consults its database.
  - The ** yes indicates that the query succeeded.

Variables

- Suppose we want to ask, "What course does Turing teach"?

  This could be written as:

  Is there a course, X, which Turing 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 said to be bound to that value.

- A variable must begin with a capital letter or "_".

- To ask Prolog to find the course which Turing teaches, type:

  `lectures(turing, Course)?`

  `Course = 9020` output from Prolog

- To ask which course(s) are taught by Prof. Codd, we may ask:

  `lectures(codd, Course)?`

  `Course = 9311`
  `Course = 9314`

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

Another example query

```
lectures(codd, 9020)?
** no
```

- If answer is yes, the query succeeded.
- If answer is no, the query failed.

- Prolog is not being intelligent about this - it would not see a difference between this query and:

```
lectures(fred, 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.

Conjunctions in Queries

- "Conjunction" means the logical operation "and".

- 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)`

  i.e. Turing lectures in a course Course and Fred studies that Course

  The question consists of two goals.

  To answer this question, Prolog must finds single value for Course that satisfies both goals.
Backtracking in Prolog 1

- Who does Codd teach?

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

  Course = 9311
  Student = jack

  Course = 9314
  Student = jill

  Course = 9314
  Student = henry

- Prolog solves this problem by proceeding left to right and then backtracking.

- Given the initial query, Prolog starts by trying to solve
  lectures(codd, Course)

  There are six lectures clauses, but only two have codd as first argument.

  Prolog chooses the first of these two clauses, i.e. lectures(codd, 9311).

  With Course = 9311, it then tries to satisfy the next goal, studies(Student, 9311).

Visualising Backtracking

- To picture what happens when Prolog tries to find a solution and backtracks, we draw a "proof tree":

Backtracking in Prolog 2

- What happens next?

  After the first solution (Course = 9311, Student = jack) is found, Prolog retracts 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 and re-binds Course to 9314.
  - goes down the lectures(codd, 9314) branch
  - tries studies(Student, 9314).
  - finds the other two solutions:
    - (Course = 9314, Student = jill)
    - (Course = 9314, Student = henry).

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:

  teaches(Teacher, Student) :- lectures(Teacher, Course), studies(Student, Course).

- 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 stands for and.

Another rule:

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

: more_advanced(henry, fred)?
: more_advanced(jack, X)?

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

Logical or in Prolog

- There are two ways to do or in Prolog; a good way and a usually bad way.
- The often bad way involves using ";". This is not taught in COMP9414 and you should not use it.
- The good way is to use two rules. For example, if I want to say that someone is happy (happy(X)) if they have lots of money (got_money(X)), or if they if they pass COMP9414 (pass_9414(X)), then I could express it like this in Prolog:

  happy(X) :- got_money(X).
  happy(X) :- pass_9414(X).

- When Prolog tries the goal happy(fred), using these rules, it first checks to see if got_money(fred) succeeds.
- If so, then happy(fred) succeeds. If not, then it tries the pass_9414 rule, and happy(fred) succeeds if pass_9414(fred) succeeds.
- The net effect is that happy(fred) succeeds if either got_money(fred) or pass_9414(fred).

Asking Questions with Structures

- How do we ask, "What books does John own that were written by someone called LeGuin"?
  owns(john, book("Tehanu", author(leguin, ursula))).

  : owns(john, book(Title, author(leguin, GivenName)))?

  Title = "Tehanu"
  GivenName = ursula

  - What books does John own?
    : owns(john, Book)?

    Book = book("Tehanu", author(leguin, ursula))

  - What books does John own?
    : 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.

Structures

<table>
<thead>
<tr>
<th>Reference:</th>
<th>Bratko chapter 2</th>
</tr>
</thead>
</table>

- 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").

  functor atom string atom

- Often 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.

  owns has arity 2. In the library database example, coming soon, book and member have arity 3, and loan has arity 4. Arity 1 is possible, too - an example would be happy(fido).
Library Database Example

• A database of books in a library contains facts of the form

book(CatalogNo, Title, author(Family, Given)).
member(MemberNo, name(Family, Given), Address).
loan(CatalogNo, MemberNo, BorrowDate, 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

• Dates are stored as structures:
  date(Year, Month, Day)

• e.g. date(2005, 6, 16) represents 16 June 2005.

• which books has a member borrowed?

borrowed(MemFamily, Title, CatalogNo) :-
  member(MemberNo, name(MemFamily, _), _),
  loan(CatalogNo, MemberNo, _, _),
  book(CatalogNo, Title, _).


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 rule uses comparison operators.

• Comparison operators are predefined in Prolog.

• More complex arithmetic expressions can be arguments of comparison operators - e.g. X + Y >= Z

Overdue Books

overdue(Today, Title, CatalogNo, MemFamily) :-
  loan(CatalogNo, MemberNo, _, DueDate),
  book(CatalogNo, Title, _),
  member(MemberNo, name(MemFamily, _), _).

Due Date

• Assume the loan period is one month:

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

• is accepts two arguments.

• The right hand argument 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 an assignment statement:
  » X is X + 1 won't work!
  » variables can only be bound once, using is or any other way

• is is not symmetric: Year2 is Year1 + 1 works but Year1 + 1 is Year2 does not

COMP9414 - Prolog - page 21

COMP9414 - Prolog - page 22

COMP9414 - Prolog - page 23

COMP9414 - Prolog - page 24
**Recursive Programs**

Reference: Bratko section 1.3 (doesn't cover trees)

**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 (this will be clarified shortly).

**Binary Trees 2**

- A structure like this could be used to represent a "binary tree":

**Recursive Programs for Recursive Structures**

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

- In Prolog we express this as:

  \[
  \begin{align*}
  \text{is_tree}(\text{empty}). & \quad \text{trivial branch} \\
  \text{is_tree}(\text{tree}(\text{Left}, \text{Data}, \text{Right})) :& - \text{is_tree}(\text{Left}), \\
  & \text{some_data(Data)}, \\
  & \text{is_tree(Right)}. & \quad \text{recursive branch}
  \end{align*}
  \]

- A non-empty tree is represented by a 3-arity term.

1 Bad style alternative: is_tree(X) :- X = empty. Urk!
Computations on Trees

- Let us define (or measure) the size of tree:

  ```prolog
tree_size(empty, 0).
  tree_size(tree(Left, _, Right),
            Total_Size) :-
    tree_size(Left, Left_Size),
    tree_size(Right, Right_Size),
    Total_Size is Left_Size + Right_Size + 1.
```

- The size of an empty tree is zero.
- The size of a non-empty tree is the size of the left sub-tree plus the size of the right sub-tree plus one for the current tree node.
- The data does not contribute to the total size of the tree.

**Notice:**

- when there is a recursive relationship between terms, we write recursive programs to describe these relationships. A recursive program is one which refers to itself, thus, `tree_size` contains goals that call `tree_size` recursively.

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.

**Reference:** Bratko chapter 3

- We could define lists as follows:

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

- A list of numbers `[1, 2, 3]` would look like:

  ```prolog
  list(1, list(2, list(3, nil)))
  ```

- Since lists are used so often, Prolog has a special notation:

  ```prolog
  [1, 2, 3] = list(1, list(2, list(3, nil)))
  ```

Examples of Lists

- `[X, Y, Z] = [1, 2, 3]`?
  This query asks Prolog to match the two terms on either side of the equals sign.

  ```prolog
  X = 1
  Y = 2
  Z = 3
  ```

- `[X | Y] = [1, 2, 3]`?
  The head and tail of a list are separated by the symbol ‘|’.

  ```prolog
  X = fred
  Y = [jim, jill, mary]
  ```

Examples of Lists 2

- The first several elements of the list can be selected before matching the tail.

  ```prolog
  [X, Y | Z] = [1, 2, 3]?
  ```

  ```prolog
  X = 1
  Y = [2, 3]
  Z = []
  ```

- `[X | Y] = [1]`?
  The empty list is written as `[]`.

  ```prolog
  X = 1
  Y = []
  ```

Lists "end" in an empty list!
More Complex List Matching

\[ [X \mid Y] = [[a, f(e)], [n, m, [2]]] \]

\[ X = [a, f(e)] \]
\[ Y = [[n, m, [2]]] \]

Notice that \( Y \) is shown with an extra pair of brackets: \( Y \) is the tail of the entire list: \([n, m, [2]]\) is just one element.

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:

  \[
  \text{member}(X, [X \mid \_]).^2 \quad \text{trivial branch}
  \]

  \[
  \text{member}(X, [_\mid Y]) :- \\
  \text{member}(X, Y). \quad \text{recursive branch}
  \]

Programming Principles for Recursive Structures

- Only deal with one element at a time.

- Believe that the recursive program you are writing has already been written.

  In the definition of \text{member}, we are already assuming that we know how to find a member in the tail.

- Write definitions, not programs!
  
  » If you are used to writing programs for conventional languages, then you are used to giving instructions on how to perform certain operations.
  
  » In Prolog, you define relationships between objects and let the system do its best to construct objects that satisfy the given relationship.

Concatenating Two Lists

- Suppose we want to take two lists, \([1, 3]\) and \([5, 2]\), and concatenate them to make \([1, 3, 5, 2]\)

- First, the trivial branch:

  \[
  \text{concat}([], L, L). 
  \]

- Next, the recursive branch:

  \[
  \text{concat}([\text{Item} \mid \text{Tail1}], L, [\text{Item} \mid \text{Tail2}]) :- \\
  \text{concat}(\text{Tail1}, L, \text{Tail2}).
  \]

- For example, consider \text{concat}([1], [2], [1, 2]):

  By the recursive branch:

  \[
  \text{concat}([1 \mid [1]], [2], [1 \mid [2]]) :- \\
  \text{concat}([1], [2], [2]).
  \]

  and \text{concat}([], [2], [2])

  holds because of the trivial branch.

- Entire program is:

  \[
  \text{concat}([], L, L). \\
  \text{concat}([\text{Item} \mid \text{Tail1}], L, [\text{Item} \mid \text{Tail2}]) :- \\
  \text{concat}(\text{Tail1}, L, \text{Tail2}).
  \]
An Application of Lists

- Find the total cost of a list of items. First, facts about costs items:
  - `cost(cornflakes, 230).`
  - `cost(cocacola, 210).`
  - `cost(chocolate, 250).`
  - `cost(crisps, 190).`
- Now, the code to compute the cost of a list:
  ```prolog
  total_cost([], 0).
  total_cost([Item|Rest], Cost) :-
      total_cost(Rest, CostOfRest),
      cost(Item, ItemCost),
      Cost is ItemCost + CostOfRest.
  ```
- Example of this code in use:
  ```prolog
  total_cost([cornflakes, crisps], X)?
  X = 420
  ```

Cut Prunes the Search

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

Cut 2

- Another example: using the `facts03` database, try
  ```prolog
  : lectures(codd, X)?
  X = 9311
  X = 9314
  : lectures(codd, X), !?
  X = 9311
  ```
- The cut in the second version of the query prevents Prolog from backtracking to find the second solution.
Controlling Execution

- Some methods for controlling execution in Prolog:
  - ordering of clauses (facts and rules)
  - ordering of goals within a rule
  - cut operator (!)

- Use each with care.

<table>
<thead>
<tr>
<th>Summary</th>
<th>Introduction to Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>We have introduced facts, queries, variables, conjunctions of goals, rules, structures, recursion, trees and lists, and controlling execution by means of the &quot;cut&quot;.</td>
<td></td>
</tr>
</tbody>
</table>