PROgramming in LOGic

- Prolog is a language which embodies the *logic programming* paradigm.

- How is the logic programming paradigm different from the imperative programming paradigm?
  - (eg what is a program? Are there variables? How are they manipulated? Are there modules? Is there branching and iteration? What data structures are there?)

see Chapter Ten

PROgramming in LOGic (2)

- Logic as the basis for a programming language is a great idea since it’s aimed at the problem, not the machine.
  - But if you’re used to imperative programming it takes a while to get used to it.

- The way that it is presented can often suggest that Prolog is just a toy, and not for serious programming. This is not so.
- It is true that Prolog is more suitable for some kinds of tasks and not others (for example?).

PROgramming in LOGic (3)

Within the CS department, Prolog has been used for:

1. Organising and printing CS department timetables. (15K)

2. Generating (and printing) the university's exam timetables. (42K)

3. Programs from specifications and proving that they are correct. (313K)

Elsewhere:

1. Implementing "business logic":

loan applications, profit evaluation, insurance evaluation

2. Clarissa: a voice interface for browsing space station procedures.

3. Natural lanaguage parsing

4. "AI" and expert systems

What is logic?

- **Logic** *n.*1. The branch of philosophy concerned with analysing the patterns of reasoning by which a conclusion is drawn from a set of premises, without reference to meaning or context

  (*Collins English Dictionary*)

Why study logic?

- Logic is concerned with two key skills, which any computer engineer or scientist should have:
  - Abstraction
  - Formalisation

Why is logic important?

- Logic is a **formalisation of reasoning**.
- Formal language for **deducing** knowledge from a small number of explicitly stated **premises** (or hypotheses, axioms, facts)
- Formal framework for **representing knowledge**
- Differentiates between the **structure** and **content** of an argument

Logic as formal language

- Logic can be presented as a **formal language**
- Within that formal language:
  - **Knowledge** can be stated concisely and precisely
  - The process of **reasoning** from that knowledge can be made rigorous

What is an argument?

- An argument is just a **sequence of statements.**
- Some of these statements, the **premises**, are assumed to be true and serve as a basis for accepting another statement of the argument, called the **conclusion.**

Deduction and inference

- If the conclusion is justified, based solely on the premises, the process of reasoning is called **deduction**
- If the validity of the conclusion is based on *generalisation* from the premises, based on strong but inconclusive evidence, the process is called **inference** (sometimes called **induction**)

Two examples

- **Deductive** argument:

  “Alexandria is a port or a holiday resort. Alexandria is not a port. Therefore, Alexandria is a holiday resort”

- **Inductive** argument

  “Most students who have blue eyes also have blonde hair. John has blue eyes. Therefore John has blonde hair.”

Some different types of logic

Propositional logic (Boole, 1815-1864)

- Simple types of statements, called **propositions**, treated as atomic building blocks for more complex statements

  Alexandria is a port or a holiday resort.

  Alexandria is not a port.

  Therefore, Alexandria is a holiday resort

- Basic propositions in the argument are

  P – Alexandria is a port

  H – Alexandria is a holiday resort.

- In abstract form, the argument becomes

  P or H

  Not P

  Therefore H

Predicate logic (Frege 1848-1925)

- Extension of propositional logic
- A ‘predicate’ is just a property
- Predicates define relationships between any number of entities using qualifiers:
  - $\forall$ “for all”, “for every”
  - $\exists$ “there exists”

THE PROLOG INTERPRETER

Prolog is more than a language: it is a language *with* an interpreter.

Prolog is interactive. The user interacts with the Prolog interpreter.

prompt:?- <goal>

Yes

or

prompt:?- <goal>

No

THE PROLOG DATABASE

- During an interactive session, the Prolog interpreter has a temporary store, called the Prolog database (just 'the database' for short). In this store it keeps *facts* and *rules*.
- When you start a session, the database is empty, and there are certain ways in which it can be added to.
- The contents of the database are the 'knowledge' that the interpreter will refer to when it tries to achieve goals.
- A 'program' in Prolog is nothing more than a set of facts and rules which can be loaded into the database.

FACTS

As the name suggests, facts represent assertions which the Prolog interpreter will regard as being *true*.

Examples of assertions:

- 'Bill is old'.
- 'Anne's age is 29'.
- 'John is the father of Mary'.
- 'Henry VIII was king of England from 1509 to 1547'.
- 'February has 28 days'.

Prolog has a way to express such assertions formally:

```
·     old(bill).
·     age(anne,29).
·     father(john,mary).
·     king(henry,8,england,1509,1547).
·     month(february,28).
```

FACTS (2)

Suppose that the fact

```
age(anne,29).
```

is in the Prolog database.

If we then give the goal

```
?- age(anne,29).
```

the interpreter will respond with Yes. (This goal is immediately achieved.)

More interestingly, if we give the goal

```
?- age(anne,A).
```

the interpreter will achieve this *by giving the value 29 to the variable A*.

Besides saying Yes, the interpreter will report that it is giving value 29 to A.

Rules

Besides simple facts, we can also store more complex information as rules. E.g.

'A person is old if he/she is over 70'.

'Date D1/M1 comes before date D2/M2 (in a given year)
if M1 < M2 *or* (M1 = M2 and D1 < D2) '.

In Prolog:

```
old(P) :- age(P,A), A > 70.
earlier(D1,M1,D2,M2) :- M1 < M2; (M1 =
M2, D1 < D2).
```

Rules(2)

A rule has a *head* and a *body*, separated by the **:-** symbol.

Three important symbols in rules:

IF is represented by the symbol **:-**

AND is represented by a comma

OR is represented by a semicolon

The *head* of a rule may not contain *AND* or *OR* symbols.

PROGRAMS

Here is a simple Prolog program:

```
    get_age :-
            write('Enter person: '),
            read(P),
            age(P,A),
            write(P), write(' has age '),
  write(A).
```

- We 'run the program' by giving `get_age` as a goal. The interpreter will find the rule, and will try to achieve the goal by achieving (separately and in order) the *subgoals* in the *body* of the rule.
- Goals involving `read` and `write` behave just as you would expect.
- The subgoal `age(P,A)` can succeed only if the database contains an appropriate fact of the form given above. If it doesn't, then `age(P,A)` will fail, and `get_age` will fail (answer `No`).

Example Database

```
    % A rule:
     get_age :-
             write('Enter person: '),
             read(P),
             age(P,A),
             write(P), write(' has age '), write(A).

% Some facts:
     age(jane,24).
     age(jill,26).
     age(julia,33).
     age(mary,29).
     age(alex,26).
     age(arthur,26).
     age(bill,33).
     age(eric,17).
     age(john,42).
```

Syntax

1. a Prolog program consists of a number of **clauses**, each ended with a **fullstop** (**.**).
2. Each clause is either a fact or a rule.
3. An **atom** is a group of alphanumeric characters starting with a **small letter**.
4. A **variable** is a group of alphanumeric characters starting with a **capital letter**.
5. Comments are enclosed by the delimiters **/*** and ***/**. Or **%**

GOALS

1. `?- 4 = 4.`
   The interpreter responds with

2. `?- 4 = 5.`
   The interpreter responds with

Note that a goal is always terminated by a *full stop*.

GOALS (2)

How can we do computations?

1. ?- X is 4 * (5 + 2).

2. ?- length([3,7,1,4],L).

OBSERVATIONS

- The interpreter knows about some things (e.g. is and length), but we shall see ways to give "teach" it more.
- On the previous slide, X and L are *variables*. In these examples the variables are given values (which the interpreter tells us). But

  Values of variables do not persist after the interpreter has reported them.

- We can give a sequence of goals (separated by commas):

```
?- X is 3 + 2, Y is 5 * 4, Z is X + Y.
```

- Here the variables X and Y carry values from the first two subgoals into the third subgoal.

- But when we give the *next* goal to the interpreter, it will have forgotten all about this X and Y.

Example

Suppose we have a file (say called `ages.plg`) containing the `age` facts that we saw before. Let's suppose this file has been consulted, by

```
?- consult(ages).
```

and consider some goals.

1. `?- age(eric,X).` (succeeds, with `X = 17`).
2. `?- age(alex,X).` (succeeds, with `X = 26`).
3. `?- age(susan,X).` (fails).
4. `?- age(P,17).` (succeeds, with `P = eric`).
5. `?- age(P,19).` (fails).
6. `?- age(P,26).` (succeeds, with `P = jill`).

Values for Variables

- In all of the above, the interpreter achieves the goal by matching the goal with a fact in the database. Generally it is this matching process which gives values to variables.

- What if there are no matches in the database?
- What if there is a possible match in the database?
- What if there are more possible matches in the database?

- Backtracking

Variables

- What is a variable?
- How is `X` a variable, but `mary` is the name of a person?

- Notes:
  - There are no declarations of variables.
  - Variables do not have types.
  - It is sensible to choose names for variables which are suggestive of their intended use.
  - Prolog is case-sensitive.
- Variables in Prolog are not global. In fact they behave more like parameters than variables.

Use Of Variables

- To provide generality in rules:
  - Suppose that we have a collection of facts of the forms

```
        father(james,mary).
        mother(jane,brian).
```

- Then it might be useful to have a rule:

```
    parent(X,Y) :-
                   father(X,Y); mother(X,Y).
```

- Here the `X` and `Y` stand for anything. Whatever `X` and `Y` stand for,

  `X` is a parent of `Y`

  IF

  `X` is the father of `Y` or `X` is the mother of `Y`

- This use of variables is similar to the use of formal parameters in a method.

Use of Variables (2)

- To pass values from one part of a computation to another:

```
    father_age(Person,Age) :-
                  father(Father,Person),
                  age(Father,Age).
```

- We might then give a goal

```
        ?- father_age(mary,A).
```

- The variable Person is being used to carry a value IN to the operation of the rule. The variable Age is carrying a value OUT.

Use of Variables (3)

- If required, this can be parcelled up with input and output:

```
get_father_age :-
                write('Enter person: '),
                read(Person),
                father_age(Person,Age),
                write('The father of '),
write(Person),
                write(' has age '),
write(Age).
```

Backtracking

- What happens in the previous example if the person entered is not known to the database (i.e. there is no father fact which can be used)?
  - Answer: the goal fails, and the interpreter just answers No.
- We can make this tidier by adding another rule:

```
get_father_age :-
              write('Enter person: '),
              read(Person),
              father_age(Person,Age),
              write('The father of '),
write(Person),
              write(' has age '), write(Age).
   get_father_age :-
              write('Not known.').
```

Backtracking (2)

- The interpreter now has another way to achieve the goal `get_father_age`.
- Initially, it will try to use the first rule. If this fails, the interpreter will automatically backtrack to try to succeed another way.

Failure always causes backtracking.

Backtracking (3)

- Consider the `age` facts again.
- Task: to write out a list of names of all the people with a given age.

```
   people_of_age(A) :-
             age(P,A),
             write(P), nl,
             fail.
     people_of_age(A) :-
             write('End of list').
```

- Two new things here:`nl` causes a new line on the output. `fail` (as a goal) always fails immediately.
- So why make it fail?

- This is our first example of a failure-driven loop.
- (But note that this looping can be caused by any failure, not just by the explicit fail.)

Backtracking (4)

- When the interpreter backtracks, it works its way back through all of the previous subgoals, trying to re-achieve each one. It is a general rule that read and write goals cannot be re-achieved, but here (possibly) age(P,A) can be.
- When some subgoal has been re-achieved, the interpreter starts working forwards again from that point in the normal way, as if nothing had happened.
- A subsequent failure will cause backtracking again.
- We have used the explicit fail here, but any failure causes backtracking.
- Note that, as a result of backtracking, variables can have their values changed (e.g. P above). This is the only way that values of variables can be changed.
- Management of backtracking is fundamental to writing programs in Prolog.

Recursion

- Recursion in Prolog means placing in the body of a rule a call to the predicate which occurs in the head of the rule.
- Here is a simple (bad) example:

```
        stars :- write('*'), nl, stars.
```

- You should try this (give the goal `?- stars`.). It doesn't work properly, and it is important to see why (and to see what happens when you try it -- CTRL-C followed by 'a' will come in handy).

Recursion (2)

- Recursion always must be made to terminate properly.

- Here is a better example:

```
stars(0) :- nl.
stars(N) :- write('*'), nl, M is N-1, stars(M).
```

- And some others (see recursion.plg):

```
    royal(victoria).
    royal(X) :- parent(P,X), royal(P).

    archer(dan).
    archer(X) :- father(P,X), archer(P).
```

Lists

- The most important data structure in Prolog is the list.
- Actual lists are normally written out like this:

```
[3,7,5,29,6,3,1,2]
[mary,john,bill,arthur]
[75]
[]
```

- But lists have alternative notations.
  [H|T] means the list with head H and tail T.
  [A,B|T] means the list whose first two members are A and B, with T as the rest of the list.

Lists (2)

- In the examples above:
  1. `3` is the head and `[7,5,29,6,3,1,2]` is the tail.
  2. `mary` is the head and `[john,bill,arthur]` is the tail.
  3. `75` is the head and `[]` is the tail.
  4. The empty list does not have a head and a tail.
  5. `[2,1,3]`,

     `[2|[1,3]]`,
     `[2,1|[3]]`,

     `[2|[1|[3|[]]]]`

     all represent the same list.

Lists (3)

- Notes:
  - The tail of a list is a list.
  - We must be careful to distinguish a one-member list from the object which is its single member.
  - Processing lists is almost always done using recursion.
  - Task: to write out the members of a list, each on a separate line.

```
writelist([]).
writelist([H|T]) :-
                  write(H), nl,
                  writelist(T).
```

- This is the first illustration of the use of patterns in the head of a rule. There are two cases here, represented by two rules, one for an empty list, and one for a list which is not empty. We shall see more of this, as it is significant.

Built-In List Operations

1. Membership of a list:

   ```
   member(X,Y).
   ```

   Succeeds when X belongs to list Y.

2. Length of a list:

   ```
   length(X,Y).
   ```

   'Y is the length of list X'.

3. Concatenate lists:

   ```
   append(X,Y,Z).
   ```

   'Z is the result of concatenating lists X and Y.'

4. Deleting from a list:

   ```
   delete(X,Y,Z).
   ```

   'Z is the result of deleting all occurrences of Y from the list X.'

5. Last in a list:

   ```
   last(X,Y).
   ```

   'X is the last member of list Y'.

Functions

- Prolog does not allow functions (methods, procedures …) .
- If we need to define a function to carry out certain computations, we can do so, but it must be done indirectly by means of a predicate.
- Example:

```
double(X,Y) :- Y is 2*X.
```

- Think of X and Y as parameters: X carries a value IN to the operation of double, and Y carries a value OUT.
- Example:

```
?- double(7,K).
```

- This will result in K being given the value 14.

Functions (2)

- Compute the larger of two numbers:

```
max(A,B,A) :- A >= B.
max(A,B,B) :- B > A.
```

- The first two parameters carry inputs, the third carries the output.

- Note that there is no explicit IF .. ELSE to distinguish the two cases. Prolog will use the first rule if it can, otherwise it will use the second.

- Example goal:

```
?- max(4,7,M).
```

- This will result in M being given the value 7.

Functions (3)

- Another:

```
                    sumlist([H|T],S) :-
                                      sumlist(T,N),
                                      S is H+N.
                    sumlist([X],X).
```

- This predicate will compute (as the value given to the second parameter) the sum of the numbers in a list (the first parameter).
- Note the use of patterns for the IN parameter. The first clause deals with lists with one member only, the second deals with lists with more than one member.
- Example goal:

```
        ?- sumlist([6,2,5,2],V).
```

- This will result in V being given the value 15.

Functions (4)

- And patterns can be used for OUT parameters, too:

```
duplicate([],[]).

duplicate([H|T],[H,H|T1]) :-

    duplicate(T,T1).
```

- This predicate will take a list (its first parameter), and duplicate every member of it, to give a new list as the value of the second parameter.

- Example goal:

```
?- duplicate([6,2,5],E).
```

- This will result in E being given the value `[6,6,2,2,5,5]`.

Structure of Programs

- How can a Prolog program be large and complex?
  - Answer: rules for predicates can call other predicates.
- Schematically, here is a more complicated program:

```
a :- b, c, d.
b.
c :- e, f.
d :- g.
e :- h.
g.
h.
```

Structure of Programs (2)

- A program is a collection of facts and rules. Normally there will be a top level predicate --- the one which will form the goal which is given to the interpreter when we 'run the program'.
- It is standard practice to place the top-level predicate at the beginning of the program. In the above, `a` is the top-level predicate.
- Programs can be spread over a number of files without difficulty. Before being run, a program must be in the database. It is loaded by using consult, and we may consult several files if we need to.

Achieving Goals

1. The interpreter is given a goal, it uses the first fact or rule in the database which matches the goal.
2. This matching may give values to variables.
3. If a fact is used, the goal is achieved immediately.
4. If a rule is used, the components of the body of the rule become subgoals, which are individually dealt with, in order.
5. Subgoals separated by commas must all be achieved.
6. Subgoals separated by semicolons are alternatives (if one fails then the next is tried).
7. If any stage causes backtracking, the interpreter tries to find other ways to achieve goals.

Readability

- It is very easy to write programs which are correct but incomprehensible.
- Comments are essential in Prolog programs, and there should always be comments to indicate, for each predicate, which other predicates call it.
- In the body of a rule, place the separate subgoals on separate lines.
- Use of blank lines, or of comment lines consisting of asterisks (say) can also help. Avoid clutter.
- Mixing up commas and semicolons in the body of a rule is a bad idea. It makes programs hard to follow, and you should avoid doing it if possible.

Example

- Recall

```
    parent(X,Y) :- father(X,Y);
     mother(X,Y).
```

- Entirely equivalent to this is the following:

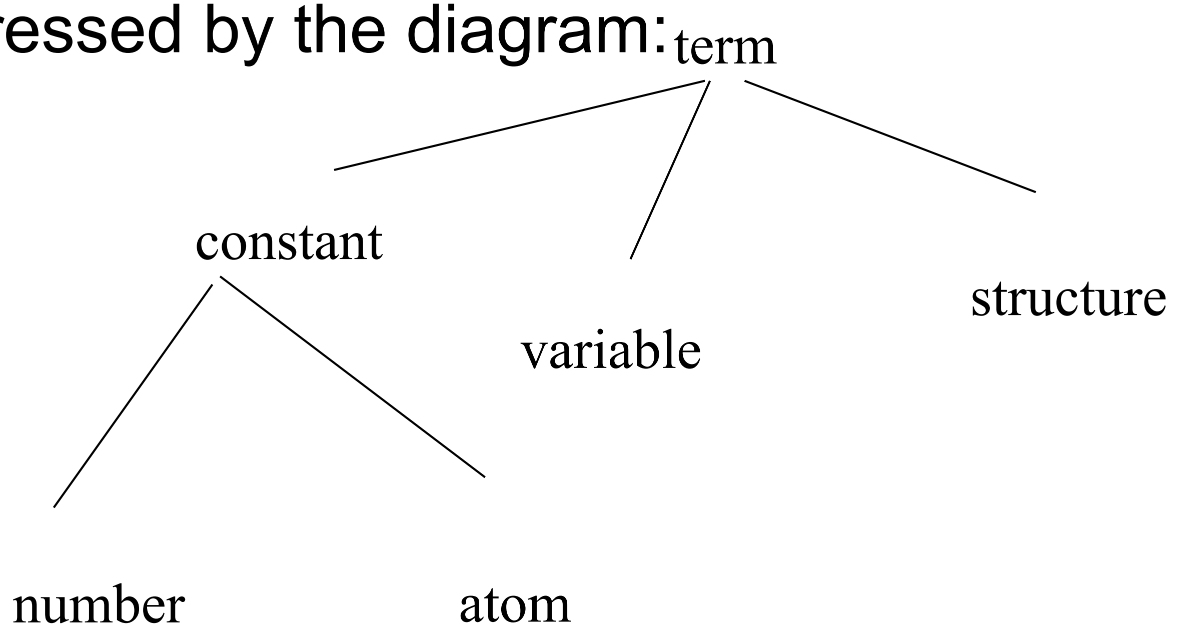
```
            parent(X,Y) :- father(X,Y).
            parent(X,Y) :- mother(X,Y).
```

I.e. two alternative rules.

- Frequently this can be done in preference to using a semicolon, and it makes programs much clearer.

Formalities

- We must spend some time describing the syntax of Prolog.
- This is in order to know what constitutes a legal expression, i.e. an expression which the Prolog interpreter will understand.
- The syntax of Prolog is very simple, and may be expressed by the diagram:



Formalities (2)

- Every legal expression is a term.
- A term is either a constant, a variable, or a structure.
- A constant is either a number or an atom.
- Now we need only describe what numbers, atoms, variables and structures are.

Syntax

- A number is just what you think.
  (Integers and floating-point numbers).

- An atom consists of characters in one of the following ways:

  1. beginning with a lower-case letter (internal underscores allowed).
  2. alphanumeric, in particular combinations which are known to the interpreter, e.g. :-, =, >=, etc.
  3. Any string of characters enclosed in single quotes.
     If you need to include a single quote in such an atom, put two single quotes, e.g.

     'Bill''s House'

Syntax (2)

- A variable is an alphanumeric string which begins with an upper-case letter or an underscore.

- A structure has a functor and arguments. The functor must be an atom, and the arguments can be Prolog terms of any kind. E.g.

```
child(mary,john)
date(29,2,96)
earlier(date(29,2,96),date(1,3,96))
```

- Every legal expression in Prolog which is not a number, an atom or a variable, is a structure with a functor and arguments.

Forms of Structures

- Some structures have alternative forms:
  1. Arithmetic expressions
  2. Lists
  3. Facts and rules.

1. Arithmetic expressions:

```
2 + 3  means  +(2,3)
2 < 3  means  <(2,3)
2 = 3  means  =(2,3)
X is 2 + 3 means  is(X,+(2,3))
```

Forms of Structures

2. Lists (the functor is always .)

```
[H|T]   means   .(H,T)
[2]     means   .(2,[])
[2,5,5] means   .(2,.(5,.(5,[])))
```

3. Facts and rules (the functor is always :-)

```
child(X,Y) :- parent(Y,X).
```

is the same as

```
:-   (child(X,Y),parent(Y,X))
```

```
age(anne,29).
```

is the same as

```
:-   (age(anne,29),true)
```

Expressions in Prolog

- Prolog can read/write an entire term in one operation.
- There are no types in Prolog. Each variable can have as its value any Prolog term.
- Numbers and structures are different things. The goal

  ```
  ?- 2 + 3 = 5.
  ```

  will fail, because 2 + 3 is a structure, whereas 5 is a number.
- To evaluate an arithmetic expression we can use **is**:

  ```
  ?- X is 2 + 3, X = 5.
  ```

  will succeed.
- There is also =:=, which evaluates then tests equality:

  ```
  ?- 2 + 3 =:= 7 - 2.
  ```

  will succeed.

Data Structures

- Strictly, the only kind of expression which can represent structured data is the structure. There are no arrays or pointers. We do have lists, of course (which are a special kind of structure).
- Simple structures, e.g.

```
father(dan,phil)
king(henry,8,england,1509,1547)
date(thursday,15,february,1996)
book('Algorithms','Sedgwick,R','Addison-
Wesley',1988)
```

- These are structures which may be considered analogous to records. Such structures may be stored as facts, and data may be extracted by the use of goals containing variables:

```
?- king(henry,8,england,X,Y).
X = 1509
Y = 1547
```

Summary

- What we've covered:
  - logic and argument
  - goals and how to achieve them
  - facts and rules
  - the Prolog database
  - predicates and programs
  - use of variables
  - backtracking
  - recursion
  - data structures including lists
  - Prolog syntax
- Prolog is different - it's a different paradigm
  - the idea of program is totally unfamiliar
  - but some ideas persist from one paradigm to another