1 Types of programming language

Imperative. Tell the computer *how* to achieve a goal, by giving a sequence of instructions.

Examples: Fortran, Pascal, Basic, Cobol.

Declarative. Tell the computer *what* goal to achieve, by giving a specification.

Examples: Prolog, OBJ3, Lisp.

Prolog and Lisp are not entirely declarative — they have imperative features too, e.g. for input and output.

2 Types of declarative language

Functional. The specification is given as a series of function definitions.

Examples: Lisp, Miranda, Hope.

Equational. The specification is given as a series of equations, or "rewrite rules".

Example: OBJ3.

Logical. The specification is given as a series of predicate definitions.

Examples: Prolog, Parlog, Trilogy.

3 An example of Prolog

We describe relations on trees. The tree is stored as a sequence of unconditional facts:

```
is_a(abel,male).
is_a(adam,male).
... etc ...

is_a(amanda,female).
is_a(anjali,female).
... etc ...

is_father_of(abel,bert).
is_father_of(abel,belinda).
... etc ...

is_mother_of(anjali,bill).
is_mother_of(anjali,bridget).
... etc ...
```

```
is_grandparent_of(X, Z) :-
    is_parent_of(X, Y),
    is_parent_of(Y, Z).


is_parent_of(X, Y) :-
    is_mother_of(X, Y).

is_parent_of(X, Y) :-
    is_father_of(X, Y).


is_half_sister_of(X, Y) :-
    is_a(X, female),
    is_parent_of(Z, X),
    is_parent_of(Z, Y),
    X \= Y.
```

4 Clauses

Prolog programs are written as a sequence of *clauses*. Each has the form

```
predicate( argument, argument, ... ).
```

or

```
predicate( argument, argument, ... ) :-
    predicate( argument, argument, ... ),
    predicate( argument, argument, ... ),
    ... .
```

`,` means "and".

`:-` means "if".

All clauses are *Horn clauses*: one goal, implied by a conjunction of other goals (possibly none).

5 Program components

Predicate names start with a lower-case letter. Example: `is_a`, `diff`.

The arguments to predicates can be any *term*:

Atoms. Arbitrary names, used as logical constants. They must start with a lower case letter.

Example: `x`, `identity_1`.

Numbers. Integers or floating-point numbers.

Example: `2`, `-25.39`.

Variables. Arbitrary names used as logical variables. They must start with a capital letter.

Example: `X`, `N_plus_1`.

Structures. An n-tuple of terms, prefixed by an atom, the *functor*.

Example: `vec3(0,1,0)`,
`mat2( vec2(1,0), vec2(0,1) )`.

Lists. Sequences. Lists are actually a type of structure, but have a special syntax.

Example: `[the,dog,bit,the,cat]`, `[]`,
`[ [1,0,0], [0,1,0], [0,0,1] ]`

6 Example of structure-processing: complex numbers.

```
add( complex(A,B), complex(C,D),
     complex(E,F)
   ) :-
    E is A+C,
    F is B+D.

multiply( complex(A,B), complex(C,D),
          complex(E,F)
        ) :-
    E is A*C - B*D,
    F is A*D + B*C.
```

Read this as:

```
The sum of (A,B) and (C,D) is (E,F) if
    E is the numerical value of A+C and
    F is the numerical value of B+D.
```

For this to work, `A`,`B`,`C`,`D` must be numbers. `is` does arithmetic. Logically speaking, `X is Y` means "X is the numerical value of the expression Y".

7 Example of list processing

```
member( X, [X|Tail] ).

member( X, [Head|Tail] ) :-
    member( X, Tail ).
```

Read this as:

```
X is a member of any list whose
first element is X.

X is a member of any list if
    X is a member of the tail of
    the list.
```

Standard terminology in list processing: The *head* of a list is its first element. The *tail* of a list is the sublist formed by its second, third, ... elements.

8 Another example of list processing

```
append( [], L, L ).

append( [H|T], L, [H|T1] ) :-
    append( T, L, T1 ).
```

Read this as:

```
The result of joining the empty list
to L is L.

The result of joining the list whose
head is H and whose tail is T to
L is a list whose head is H and whose
tail is T1 if
    the result of joining T to L is T1.
```

9 Uses of lists

Lists can be used to represent sets. They can also be used to represent trees:

```
[ plus, a, b ].

[ plus, [times,a,b], [times,c,d] ].
```

Above, we use them to represent expressions built up from the operators `plus` and `times`. If these operators are commutative, it is often convenient to give them more than two arguments:

```
[ plus, a, b, c, d ].
```

instead of

```
[ plus, [ plus, a, b ], [ plus, c, d ] ].
```

Many algebra-manipulation programs use this method.

10 Lists for parse trees

In linguistics, lists can represent parse trees:

```
[ sentence,
    [ noun_phrase,
      [ determiner,
        [the]
      ],
      [ noun,
        [dog]
      ]
    ],
    [ verb_phrase,
      [ verb,
        [hates]
      ]
    ],
    [ noun_phrase,
      [ noun,
        [cats]
      ]
    ]
]
```

We can use the same representation when parsing formal languages, such as programming languages.

11 Lists for graphs

By putting "labels" inside lists, we can represent cyclic graphs:

```
[ label(1,a), b, c, ref(1) ].
```

It may be more convenient though, to represent the edge-set:

```
[ edge(a,b), edge(b,c), edge(c,a) ].
```

Doing this, we can associate weights and other information with the edges:

```
[ edge(a,b,0.3), edge(b,c,0.4),
  edge(c,a,27) ].
```

This is useful in handling finite-state automata:

```
[ arc(s1,s2,i1), arc(s1,s3,i2),
  arc(s3,s1,i3), arc(s3,s3,i4) ].
```

We could use a similar representation for diagrams in category theory.

12 The two ways to represent graphs

So there are two approaches to representing trees and other graphs:

- As sequences of clauses.
- As lists or other structures.

The first is better when the program is working with one graph which remains unchanged throughout. For instance, a program to simulate a network of computers. This could be given a set of clauses defining the layout of the network, and then work on that.

The second is better when the program needs to create lots of new graphs. For example, programs for grammatical analysis, or for reducing finite-state machines, or for pasting categorical diagrams.

13 Operators

In the structure `complex(a,b)`, `complex` is called the *functor*. Normally, structures are written in prefix notation, with the functor before its arguments.

Prolog allows us to declare that a given functor can be written between two arguments (infix), after an argument (postfix), or before an argument (prefix). Doing this avoids the need for brackets. Such functors are called *operators*. Operators can be given different precedences and associativities.

Some functors, such as `+`, `*`, `is`, and `\=`, are already defined as operators by the system. This is why we can write `C is A+B` instead of `is(C,+(A,B))`. However, the bracketed form is also allowed.

Note: These operators are atoms. As well as sequences of letters, atoms can be sequences of "symbol characters". Examples: `+`, `++`, `-->`.

14 Symbolic differentiation

Using operators allows us to write very readable programs. Example: symbolic differentiation.

```
d( X, X, 1 ).

d( U+V, X, A+B ) :-
   d( U, X, A ),
   d( V, X, B ).

d( U*V, X, B*U+A*V ) :-
   d( U, X, A ),
   d( V, X, B ).
```

Read the second clause as

```
The differential of U+V with respect
to X is A+B if
    the differential of U with respect
    to X is A,
    and the differential of V with
    respect to X is B.
```

How much nicer this is than coding the structures in C or Pascal!

15 A note on the quantification of variables

Like lists, clauses have two parts, *head* and *tail*.

```
Head :- Tail.

Head.
```

In the second clause, the tail is assumed to be `true`.

Variables which appear in the head of the clause are universally quantified.

```
d( U+V, X, A+B ) :-
   d( U, X, A ),
   d( V, X, B ).
```

Variables which appear only in the tail of the clause are existentially quantified.

```
is_grandparent_of(X, Z) :-
    is_parent_of(X, Y),
    is_parent_of(Y, Z).
```

Read this as

```
X is a grandparent of Z if
    there is some Y for which
    X is a parent of Y and
    Y is a parent of Z.
```

These are the only quantifications possible. You can't write explicit quantifiers into a clause (though they can sometimes be simulated with various programming tricks).

16 The top-level interpreter

I have shown lots of definitions. How can you use them?

All Prolog systems come with a *top-level interpreter*. This is a program which reads *questions* from the user, and calls Prolog's inference mechanism to determine their truth.

```
Loop:
    Display a '?-' prompt.
    Read the next question.
    If it is true,
        display the variable bindings
    otherwise
        say 'no'
    endif.
    Goto Loop.
```

17 Example of top-level interpreter

Example, using the simple differentiator in SYMDIFF.PL:

```
?- d( x^2, x, Result ).
Result = 2 * 1 * x ^ (2 - 1)
```

`d` does not simplify its result. The predicate `simp` does, and we can combine them as if in the tail of a clause:

```
?- d( x^2, x, Result ), simp( Result, SimplifiedResult ).
Result = 2 * 1 * x ^ (2 - 1)
SimplifiedResult = 2 * x
```

All the variables that appear in questions are existentially quantified.

18 Non-logical predicates

We don't always want to keep typing into the top-level interpreter. How can we implement other forms of interaction?

Prolog has a number of *built-in predicates*. You have seen `is` and `\=`. There are also predicates for input and output. These belong to the *non-logical* part of Prolog: they have no logical interpretation because they change the world. To use them, you have to know how Prolog is executed. Note: Prolog programmers always call them predicates, even though they have no logical interpretation.

19 Example of non-logical predicates

```
happy_birthday :-
    read(Name),
    write('Happy Birthday '),
    write(Name),
    write(' '),
    nl.
```

Read this as:

```
To do happy_birthday:
    Read a name from the terminal;
    write 'Happy Birthday ';
    write the name;
    write one space;
    move to a new line.
```

Prolog always executes the tail of a clause from left to right, as in conventional programming languages.

20 Other built-in predicates

Most Prolog systems will implement other built-in predicates for formatted input and output, for handling windows, and so on. In addition, most Prologs can now call procedures written in C and other languages, giving you access to their input and output (as well as to other libraries such as NAG).

Prolog also has built-in predicates for reading files of clauses into the database, and for adding or deleting clauses as the program runs.

21 When dynamically adding clauses is useful

Consider this definition of Fibonacci numbers:

```
fib( 0, 1 ).

fib( 1, 1 ).

fib( N, F ) :-
    N1 is N - 1,
    N2 is N - 2,
    fib( N1, F1 ),
    fib( N2, F2 ),
    F is F1 + F2.
```

Each time `fib` is called, it will work right down to `fib(0)`. This means that it wastes its time in repeatedly recalculating results.

22 Memo-functions

We can avoid this by using *memo-functions*:

```
fib( N, F ) :-
    memo_fib( N, F ).

fib( 0, 1 ).

fib( 1, 1 ).

fib( N, F ) :-
    N1 is N - 1,
    N2 is N - 2,
    fib( N1, F1 ),
    fib( N2, F2 ),
    F is F1 + F2,
    asserta( memo_fib(N,F) ).
```

Clauses for `memo_fib` get generated as the program runs, and avoid recalculating already-known results.

23 Memo-functions with lists

However, asserting new clauses is rather slow in some Prolog systems. So instead, we can pass the newly-generated results around in a list:

```
fib( N, F ) :-
    fib( N, F, [ done(0,1), done(1,1) ], _ ).
```

```
fib( N, F, L, L ) :-
    member( done(N,F), L ).

fib( N, F, L, [ done(N,F) | L2 ] ) :-
    N1 is N - 1,
    N2 is N - 2,
    fib( N1, F1, L, L1 ),
    fib( N2, F2, L1, L2 ),
    F is F1 + F2.
```

Read `fib(N,F,L,L1)` as the following relation between N, F, L, and $L1$:

(1) $\texttt{fib}(N) = F$.
(2) $L1 \;=\; \{\; (N', \texttt{fib}(N')) \;\mid\; N' \;\epsilon\; \{1 \;\ldots\; N\;\}\;\}$.
(3) $L \;\subset\; L1$.

This technique is often used in processing graphs, where we need to record the points we have already processed, in order to avoid processing them again.

There are more efficient ways to represent sets than as unordered lists. Other possibilities: ordered lists, binary trees. See books on data structures in the computing science literature.

24 How Prolog works

Prolog is not a general-purpose theorem prover, and its powers of inference are limited. For example, it cannot answer these questions:

```
X > 1, X =< 2, integer(X).
25 is X^2 + Y^2, integer(X), integer(Y).
```

Prolog works by *backward-chaining*, using Modus Ponens:

```
If I am asked to prove Q
and I have the rule (P implies Q)
then if I can prove P, I can prove Q
    otherwise I shall assume Q
    is false.
```

Note that if Prolog can't prove a goal, it assumes it to be false. This is called the *closed world assumption.*

I shall illustrate execution using the following set of rules.

```
/*10*/   suits(Skier, st_sartre) :-
            is_a(Skier, beginner),
            wants(Skier, fun).
```

```
/*20*/   suits(Skier,
               schloss_heidegger) :-
            is_a(Skier, beginner),
            wants(Skier, serious).

/*30*/   suits(Skier, chateau_derrida) :-
            is_a(Skier, advanced),
            wants(Skier, serious).

/*40*/   suits(Skier,
               wittgenstein_gladbach) :-
            is_a(Skier, advanced),
            wants(Skier, fun).

/*50*/   is_a(Skier, beginner) :-
            had_lessons(Skier, L),
            L < 30.

/*60*/   is_a(Skier, beginner) :-
            had_lessons(Skier, L),
            L >= 30,
            has_fitness(Skier, poor).

/*70*/   is_a(Skier, advanced) :-
            had_lessons(Skier, L),
            L >= 30,
            has_fitness(Skier, good).

/*80*/   has_fitness(Skier, poor) :-
            max_pressups(Skier, P),
            P < 10.

/*90*/   has_fitness(Skier, good) :-
            max_pressups(Skier, P),
            P >= 10.

wants(eddie,fun).
max_pressups(eddie,170).
had_lessons(eddie,78).
```

25 The effect of clause ordering

Because Prolog searches for clauses from top to bottom, the order in which they are written makes a difference. If a predicate has several solutions, their order of appearence is determined by the order of the clauses.

If the conditions in alternative clauses are not mutually exclusive, then backtracking will produce them both:

```
has_fitness(Skier, poor) :-
    max_pressups(Skier, P),
    P < 15.

has_fitness(Skier, good) :-
    max_pressups(Skier, P),
    P >= 10.

max_pressups(eddie,12).
```

26 The cut

To avoid writing the negation of a condition, Prolog uses the *cut*, written as `!`:

```
has_fitness(Skier, poor) :-
    max_pressups(Skier, P),
    P < 15,
    !.

has_fitness(Skier, good).

max_pressups(eddie,12).
```

The `!` in the first clause means "if you get this far, then discard (cut out) any alternative solutions".

Cut is very frequently used. But it destroys the logical reading of clauses which contain it. Logically, the clauses above mean

```
Skier has fitness poor if
    he can do less than 15 press-ups.

Skier has fitness good.
```

The second sentence contradicts the first.

27 Advantages of Prolog

As I said above, Prolog is not a general-purpose theorem prover. It is a programming language in which procedures can be written as logical specifications. You have to know how it executes predicates in order to use it profitably.

Why use Prolog rather than C?

- It has built-in list handling, very useful for representing sequences, trees, and so on.

- It is easy to read and write programs which build structures.

  ```
  add( complex(A,B), complex(C,D),
       complex(E,F)
     ) :-
      E is A+C,
      F is B+D.
  ```

  Most Prologs nowadays should be able to handle structures of at least 30Kbytes in size.

- Although Prolog is not a complete implementation of logic, it is much closer to it than C, and is more like mathematical notation.

- It is easy to build tables and databases while a program is running, as I did with `memo_fib`. You don't need a lot of programming effort.

- You can reason about programs as algebraic objects. When writing a predicate, begin with a definition that is obviously correct, but inefficient. Then apply transformations that preserve its effect while making it more efficient. There is an example in REPLACE.PL. See also *The Craft of Prolog* by Richard O'Keefe (MIT Press), and *Essentials of Logic Programming* by C.J.Hogger (Oxford University Press). I highly recommend O'Keefe.

  You can also use familiar methods of reasoning such as induction to prove that programs are correct. You may need to take the execution mechanism into account, but much less so than you would in C. See the proof in INTERVALS.PL.

28 What are Prolog's disadvantages?

- It tempts you to write things that look logically correct, but that won't run.

- The obvious way to write a predicate is unlikely to be efficient. You must know when a predicate needs to be optimised.

- Because it lacks functional notation, predicates can become cumbersome.

```
f( N, F ) :-
    N1 is N - 1,
    f( N1, F1 ),
    F is N * F1.
```

It would be much nicer to write

```
f( N ) = N * f(N-1).
```

- Input and output is not always easy.
- There are some features which have not been standardised, and differ between implementations. For example: formatted input and output, file-handling, sorting predicates.
- You can't re-assign to parts of data structures. This makes it impossible to implement arrays. However, functional programmers have developed a number of fast-access data structures which do almost as good a job.

29 Techniques

In the rest of this lecture, I shall talk about two techniques:

- Incorporating strategic knowledge.
- Implementing other inference systems.

30 Incorporating strategic knowledge

The symbolic differentiator was easy to write because the differential of an expression is easily constructed by combining the differentials of its sub-expressions. It is always obvious how to split the expression: at the principal operator. Many symbolic problems, such as integration, do not have this property. One may need to try a lot of alternative decompositions before finding the right ones. If we wrote the rules for integration in the same way as we did for differentiation, Prolog would eventually find an answer, but only after much searching.

To improve performance, we can incorporate *strategic knowledge*, to guide the program in finding the most appropriate rule. I shall illustrate this in equation-solving, for example transforming $x = \cos(y/2)$ to $y = 2\arccos(x)$.

Successful human equation do not apply the axioms of algebra blindly. Instead, they have learnt various strategies. *PRESS* was a program, written by Alan Bundy of Edinburgh University, for solving the equations found in applied mechanics problems from English A-level questions. It was part of a bigger program called MECHO, for reading and answering these questions. In writing it, Bundy incorporated knowledge gleaned by observing human mathematicians. See "The Computer Modelling of Mathematical Reasoning" by Bundy.

31 PRESS

How PRESS worked:

Consider the following stages in solving
$\log(x+1) + \log(x-1) = c$.

1. $\log(x+1) + \log(x-1) = c$.
2. $\log(x+1)(x-1) = c$.
3. $\log(x^2-1) = c$.
4. $x^2 - 1 = e^c$.
5. $x^2 = e^c + 1$.
6. $x = \pm\sqrt{e^c+1}$.

In going from (3) to (6), we "unwrap" the x at each stage, removing the dominant function on the left-hand side, and applying its inverse to the right-hand side. This method — Bundy calls it *isolation* — gradually decreases the "depth" of x until it is completely isolated. It will always work if there is only one occurrence of x, and the functions have inverses.

In going from (2) to (3), we reduce the number of x's. This — *collection* — is a necessary preliminary to isolation. We have to keep collecting x's until we only have one.

In going from (1) to (2), we can't collect the x's; they aren't close enough. Instead, we "move them closer together". Bundy calls this *attraction*.

So PRESS has the following structure:

```
solution( Lhs=Rhs, SolvedEquation ) :-
    occurrences_of( x, Lhs, 1 ),
    occurrences_of( x, Rhs, 0 ),
    isolate( Lhs=Rhs, SolvedEquation ).

solution( Equation, SolvedEquation ) :-
    occurrences_of( x, Equation, N ),
    N > 1,
    collect( Equation, Equation1 ),
    solution( Equation1, SolvedEquation ).

solution( Equation, SolvedEquation ) :-
    occurrences_of( x, Equation, N ),
    N > 1,
    attract( Equation, Equation1 ),
    solution( Equation1, SolvedEquation ).
```

Each of `attract`, `collect`, and `isolate` tries relevant algebraic transformations. For example:

```
attract( Equation, Equation1 ) :-
    sub_expression( E, Equation ),
    attract_rewrite( E, E1 ),
    replace_sub_expression( E, Equation, E1, Equation1 ).

attract_rewrite( log(U)+log(V), log(U*V) ) :-
    occurrences_of( x, U, NU ),
    NU > 0,
    occurrences_of( x, V, NV ),
    NV > 0.


collect( Equation, Equation1 ) :-
    sub_expression( E, Equation ),
    collect_rewrite( E, E1 ),
    replace_sub_expression( E, Equation, E1, Equation1 ).

collect_rewrite( (x+N)*(x-N), (x^2-N_squared) ) :-
    N_squared is N^2.


isolate( x=Rhs, x=Rhs ) :- !.

isolate( Equation, SolvedEquation ) :-
    isolate_rewrite( Equation, Equation1 ),
    isolate( Equation1, SolvedEquation ).

isolate_rewrite( log(A)=B, A=exp(B) ).
isolate_rewrite( A-X=B, A=X+B ).
isolate_rewrite( A^N=B, A=B^(1/N) ).
```

32 Implementing other inference systems

We have shown how structures can represent graphs, sets, parse trees, and equations. We can also represent inference rules as structures, and write our own inference program (often called an *inference engine*) that operates on them. This way, we can implement

- Different logical systems from Prolog.
- The same logical system, but with a different execution strategy. For example, it might try all clauses in parallel, trying goal 1 of the first, then goal 1 of the second, and so on.

Examples are given in the programs UNCERTAIN.PL and BITSTREAM.PL.