1 Introduction: Why Learn Prolog?

Prolog is PROgramming in LOGic

Prolog is based on First Order Predicate Logic.

Predicate Logic in that it has a set of predicate symbols.

First Order in that there is no means of reasoning provided for "talking about" the predicates themselves.

Prolog was originally developed at the University of Marseilles by Alain Colmerauer and his group about 1970.

Prolog has rapidly gained in popularity over the last five years. The current interest in Prolog partly stems from the Japanese Fifth Generation Project's decision to adopt DEC-10 Prolog as the core language which was to be developed further to meet their own requirements.

1.1 What Is So Useful About Programming In Logic?

A reasonable number of applications have been found. These include:

Automatic Theorem Proving
Program Verification
Program Transformation
Planning
Compiler Writing
Intelligent Knowledge Based Systems
Natural Language Processing
Expert Systems

1.2 Declarative Vs Procedural Programming

Procedural programming requires that the programmer tell the computer what to do. That is, how to get the output for the range of required inputs. The programmer must know an appropriate algorithm.

Declarative programming requires a more descriptive style. The programmer must know what relationships hold between various entities.

1.3 Program Verification

[illegible] we guarantee that a completed program is actually correct. That is, [illegible]ogram produces the desired (and only the desired) responses. This [illegible] the existence of some specification of the desired behaviour.

[illegible]i write the program specification in full first order predicate Logic. [illegible] will find a Logic Interpreter which executes the specification. There is a proviso: we still need to guarantee that the specification is not faulty.

Unfortunately, Pure Prolog uses Horn Clause notation which is not as expressive as full first order predicate logic.

1.4 Program Synthesis - Simple View

We write the specification in Horn Clause form. The specification is immediately executable but inefficient. We transform the program to a more efficient form.

1.5 Prolog Is An "AI" Language

Prolog is an "Artificial Intelligence" programming language. Up until recently, if you used Prolog then, generally speaking, you were writing an AI (Artificial Intelligence) program.

What features must an "AI" language have? There are two types of answer: one describing the features required that are not built into such languages as BASIC, FORTRAN, ALGOL, PASCAL and the other describing the features required to do the work that interests AI researchers.

1.6 Programs As Data

It is believed that one of the requirements of intelligent behaviour is the ability to reflect on one's own thinking.

By an analogy, we may reasonably require that, for Artificial Intelligence research, we need programming languages that can be used to reason about programs. Prolog permits the programmer to do this.

1.7 The Prolog Course: What You Will Learn

You are expected to learn about:

The Syntax of Prolog
The Declarative Semantics of Prolog
The Procedural Semantics of Prolog

also something about

Search Strategies
Inference
Pattern Matching
Parsing using Prolog

1.8 Organisation Details

This series of lectures is designed to provide an informal introduction to the remainder of the course.

In all, there will be twelve lectures (including this one), three tutorials and three practical sessions.

As you probably know, towards the end of the year you will be expected to choose a project on which to work which will be written in Prolog.

2 Knowledge Representation

2.1 Propositional Calculus

Based on statements which have truth values (True or False).

The calculus determines the truth values associated with certain statements formed from "atomic" statements. That is:

If p stands for "fred is rich" and q for "fred is tall" then we may form statements such as:

p or q
p and q
p logically implies q
p is logically equivalent to q
not p

The Problem

If p stands for "All dogs are smelly" then we cannot prove that "my dog is smelly".

We need to be able to get at the structure and meaning of statements.

2.2 First Order Predicate Calculus

If "The capital of France is Paris" we can represent this in Predicate Calculus form as:

```
france has_capital paris
```

We have a binary relationship (two things are related) written in infix form. The relationship (or predicate) has been given the name "has_capital" -hence we say that the predicate name is "has_capital".

and in Prolog form by such as:

```
has_capital(france,paris).
```

where we write a prefix form and say that the relationship takes two arguments.

Note that, if the name of an object starts with a lower case letter then we refer to a specific object. Also, there must be no space between the predicate name and the left bracket "(". The whole thing must also end in a ".".

2.3 Prolog Atoms

If we have

```
loves(jane,jim).
```

then "jane" and "jim" are atoms. In each case, we refer to a specific object. Also, "loves" happens to be an atom too because it refers to a specific relationship.

Because Prolog is modelled on first order predicate logic all predicate names must be atoms.

There are other atoms -including integers, real numbers and certain other entities.

2.4 Goals and Clauses

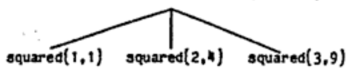
loves(jane,jim)	is a goal
loves(jane,jim).	is a (unit) clause

2.5 Disjunctions

A predicate may be defined by a set of clauses with the same predicate name and the same number of arguments.

```
squared(1,1).
squared(2,4).
squared(3,9).
```

Here is the same written as a (sort of) OR tree



2.6 Rules

The format is:

```
divisible_by_two(X):-
        even(X).
```

This is a (Non Unit) Clause.

The head is "divisible_by_two(X)" and the body is "even(X)". "even(X)" is sometimes referred to as a subgoal.

Only one goal is allowed in the head. Any number of subgoals may be in the body of the rule.

2.7 Semantics

Here is an informal version of the procedural semantics for the example above:

If we can find a value of X that satisfies the goal "even(X)" then we have found a number that satisfies the goal "divisible_by_two(X)".

The declarative semantics.

If we can prove that X is "even" then we have proved that X is "divisible_by_two".

2.8 The Logical Variable

If an object is referred to by a name starting with a capital letter then the object has the status of a logical variable. In the above rule there are two references to X. All this means is that the two references are to the same object -whatever that object is.

The scope rule for Prolog is that two uses of an identical name for a logical variable only refer to the same object if the uses are within a single clause. Therefore in

3 Prolog's Search Strategy

3.1 Queries and Disjunctions

When Prolog is entered we are at top level. As, at this level, Prolog normally expects queries it prints the prompt:

```
?-
```

and expects you to type in a single goal. Perhaps we would like to determine whether or not

```
woman(jane)
```

In this case we would type this in and see (your typing underlined):

```
?- woman(jane).
```

Now "?- woman(jane)." is also a clause. Essentially, a clause with an empty head.

We now have to find out "if jean is happy". To do this we must search through the facts and rules known by Prolog to see if we can find out whether this is so. Here are some facts:

```
woman(jean).
man(fred).
woman(jane).
woman(joan).
woman(pat).
```

In order to solve this goal Prolog is confronted with a search problem which is trivial in this case. How should Prolog search through the set of (disjunctive) clauses to find that it is the case that "jane is a woman"?

Such a question is irrelevant at the level of predicate calculus. We just do not want to know how things are done. It is sufficient that Prolog can find a solution. Nevertheless, Prolog is not pure first order predicate calculus so we think it important that you face up to this fairly early on.

The answer is simple. Prolog searches through the set of clauses in the same way that we read (in the west). That is, from top to bottom. First, Prolog examines

```
woman(jean).
```

and finds that

```
woman(jane).
```

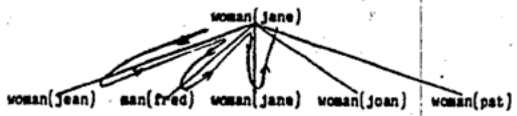
does not match.

This is fairly obvious to us! Also, it is obvious that the next clause "man(fred)." doesn't match either. >From now on we will never consider matching clauses whose predicate names (and arities) differ.

Prolog then comes to look at the third clause and it finds what we want. All we see (for the whole of our activity) is:

```
?- woman(jane).

yes

?-
```

Now think about how the search space might appear using the AND/OR tree representation. The tree might look like:



We see that the search would zig zag across the page from left to right stopping when we find the solution.

3.2 A Simple Conjunction

Now to look at a goal which requires the solving of two subgoals. Here is our set of facts and one rule.

```
woman(jean).
man(fred).
wealthy(fred).
happy(Person):-
        woman(Person),
        wealthy(Person).
```

We shall ask whether "jean is happy". We get this terminal interaction:

```
?- happy(jean).

no

?-
```

Now why is this the case? We said that we would not bother with clauses with differing predicate names. Prolog therefore has only one choice —to try using the single rule. It has to match:

```
happy(joan)
```

against

```
happy(Person)
```

We call this matching process unification. What happens here is that the logical variable "Person" gets bound to the atom "jean". You could paraphrase "bound" as "is temporarily identified with".

To solve our problem, Prolog must set up two subgoals. But we must make sure that, since "Person" is a logical variable, that everywhere in the rule that "Person" occurs we will replace "Person" by "jean". We now have:

```
happy(joan):-
        woman(jean),
        wealthy(jean).
```

```
happy(X):-
        healthy(X).
wise(X):-
        old(X).
```

the two mentions of X do not necessarily refer to the same object.

2.9 Rules and Conjunctions

A man is happy if he is rich and famous

translates to:

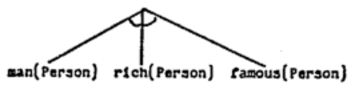
```
happy(Person):-
        man(Person),
        rich(Person),
        famous(Person).
```

The whole of the above is a (non unit) single clause.

It has three subgoals in its body.

The logical variable "Person" refers to the same object.

Here is an AND tree that represents the above.



2.10 Rules and Disjunctions

You are happy if you are healthy, wealthy or wise.

translates to:

```
happy(Person):-
        healthy(Person).
happy(Person):-
        wealthy(Person).
happy(Person):-
        wise(Person).
```

The predicate name "happy" is known as a functor.

The functor "happy" has one argument.

It has one argument —we say its arity is 1.

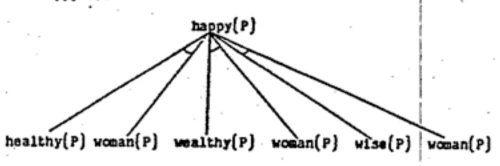
The predicate "happy/1" is defined by thee clauses.

2.11 Both Disjunctions and Conjunctions

```
happy(Person):-
        healthy(Person),woman(Person).
happy(Person):-
        wealthy(Person),woman(Person).
happy(Person):-
```

```
        wise(Person),woman(Person).
```

Again, we can form an AND/OR tree which shows the structure of the definition of "happy/1".



So the two subgoals are:

```
    woman(jean)
    wealthy(jean)
```

Here we come to our next problem. In which order should Prolog try to solve these subgoals? Of course, in predicate logic, there should be no need to worry about the order. It makes no difference -therefore we should not need to know how Prolog does the searching.

Prolog is not quite first order logic yet. So we will eventually need to know what goes on. The answer is that the standard way to choose the subgoal to work on first is again based on the way we read (in the west)! We try to solve the subgoal "woman(jean)" and then the subgoal "wealthy(jean)".

There is only one possible match for "woman(jean)": our subgoal is successful. However, we are not finished until we can find out if "wealthy(jean)".

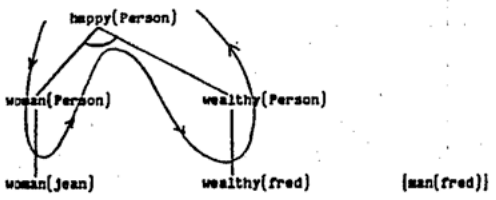
There is a possible match but we cannot unify

```
    wealthy(fred)
```

with

```
    wealthy(jean)
```

So Prolog cannot solve our top level goal -and reports this back to us. Things would be much more complicated if there were any other possible matches. Now to look at the AND/OR tree representation of the search space. Here it is:



[illegible]hat it becomes very clear that knowing that "fred [illegible]n" is not going to be of any use.

[illegible]ee that the way Prolog searches the tree for AND choices is to zig [illegible] left to right across the page! This is a bit like how it processes the [illegible]hoices except that Prolog must satisfy all the AND choices at a node before going on.

3.3 Conjunctions and Disjunctions

We are now ready for the whole thing: let us go back to the original set of facts and rules.

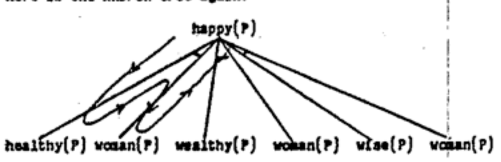
```
    woman(jean).
    woman(jane).
    woman(joan).
```

```
    woman(pat).

    wise(jean).

    wealthy(jane).
    wealthy(jim).

    healthy(jim).
    healthy(jane).
    healthy(jean).
```

and consider the solution of the goal

```
    happy(jean)
```

here is the AND/OR tree again:



and the goal succeeds.

Note that

1. Both the subgoal healthy(jean) and woman(jean) have to succeed.

2. We then return to the top level.

Now consider the top level goal of

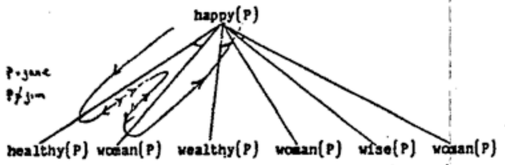
```
    happy(joan).
```

After failing healthy(joan) Prolog does not try to solve woman(joan). It goes on to try to solve wealthy(joan) -which fails. Next, Prolog tries wise(joan) -and fails. Now back to top level to report the failure to satisfy the goal.

Now consider

```
    happy(P)
```

as the top level goal.



Much more complicated. First, healthy(P) succeeds binding P to jim but when

the conjunction "woman(jim)" is attempted it fails. Prolog now backtracks.

It reverses along the path through the tree until it can find a place where there was an alternative solution.

Of course, Prolog remembers to unbind any variables at the places in the tree where they were bound.

In the example we are using we try to resolve the goal "healthy(P)" -succeeding with P bound to jane. Now the conjunction can be satisfied as we h[illegible] [illegible]man(jane)". Return to top level with P bound to jane to report [illegible] What follows is what appears on the screen:

```
    [illegible]ppy(P).

    [illegible]ne
    yes
```

Basically, trying to follow the behaviour of Prolog around the text of the program can be very messy. seeing how Prolog might execute the search is much more coherent but it requires some effort before getting the benefit.

4 Recursion, Lists and Unification

4.1 Recursion

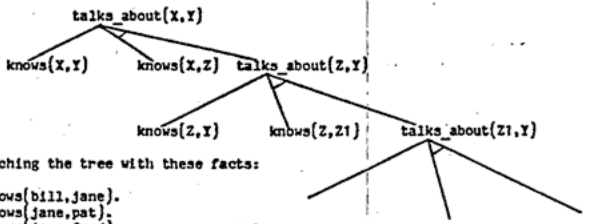
An example recursive program:

```
    talks_about(A,B):-
            knows(A,B).
    talks_about(P,R):-
            knows(P,Q),
            talks_about(Q,R).
```

If you look at the AND/OR tree of the search space you can see that

a) There is a subtree which is the same shape as the whole tree reflecting the single recursive call to talks_about.

b) The solution of a given problem depends on being able to stop recursing at some point. Because the leftmost path down the tree is not infinite in length it is reasonable to hope for a solution.



In searching the tree with these facts:

```
    knows(bill,jane).
    knows(jane,pat).
    knows(jane,fred).
    knows(fred,bill).
```

using the goal

```
    talks_about(X,Y)
```

If we ask for repeated solutions to this goal, we get, in the order shown,

```
    X= bill  Y= jane
    X= jane  Y= pat
    X= jane  Y= fred
    X= fred  Y= bill
    X= bill  Y= pat
    and so on
```

The search strategy implies that Prolog keep on trying to satisfy the subgoal "knows(X,Y)" until there are no more solutions to this. Prolog then finds that, in the second clause for talks_about, it can satisfy the "talks_about(X,Y)" goal by first finding a third party who X knows. It satisfies "knows(X,Z)" with X=bill, Z=jane and then recurses looking for a solution to the goal "talks_about(jane,Z)". It finds the solution by matching against the second knows clause.

The above AND/OR tree was formed by taking the top level goal and, for each clause with the same predicate name and arity, creating an OR choice leading to subgoals constructed from the bodies of the matched clauses. For each subgoal in a conjunction of subgoals we create an AND choice.

Note that we have picked up certain relationships holding between the (logical) variables but we have had to do some renaming to distinguish between attempts to solve subgoals of the form "talks_about(A,B)" recursively.

4.2 Lists

a) How to construct/deconstruct a list

```
    [X|Y] = [f,r,e,d]
```

will result in

```
    X= f
```

-the first element of the list is known as the HEAD

```
    Y=[r,e,d]
```

-the list formed by deleting the head is the TAIL

b) The Empty List

Simply written

```
    []
```

c) Some Possible Matches

```
    1. [b,a,d]=[d,a,b]       fails
    2. [X]=[b,a,d]           fails
    3. [X|Y]=[he,is,a,cat]   succeeds with
                             X=he, Y=[is,a,cat]
    4. [X,Y|Z]=[a,b,c,d]     succeeds with
                             X=a, Y=b, Z=[c,d]
    5. [X|Y]=[]              fails -the empty list
                             can't be deconstructed
    6. [X|Y]=[[a,[b,c]],d]   succeeds with
                             X=[a,[b,c]], Y=[d]
```

d) A Recursive Program Using Lists

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

4.3 Unification

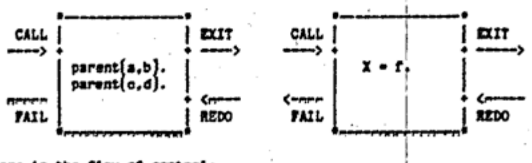
Unification is the name given to the way Prolog does its matching. The infix predicate =/2 tries to unify both its arguments. Here are some possible unifications

```
    1. X=fred        succeeds
    2. jane=fred     fails because you can't
                     match two distinct atoms
    3. Y=fred,X=Y    succeeds with X=fred, Y=fred

    4. X=happy(jim) succeeds
```

5 The Box Model of Execution and Lists

5.1 The Box Model -An Example

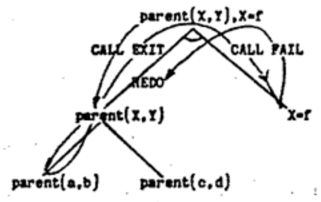
```
    ?- parent(X,Y),X=f.
```



Here is the flow of control:

```
   CALL: parent(X,Y)
                          EXIT: parent(a,b)

                                      CALL: a=f
                                      FAIL: a=f

                          REDO: parent(X,Y)
                          EXIT: parent(c,d)

                                      CALL: c=f
                                      FAIL: c=f

                          REDO: parent(X,Y)
  FAIL: parent(X,Y)
```

Below, we have a snapshot of how the execution takes place "taken" at the moment when control forces an attempt to backtrack to find another solution to the goal "parent(X,Y)".



5.2 List Processing

5.2.1 Program Patterns

a) Test for Existence

```
    nested_list([Head|Tail]):-
            sublist(Head).
    nested_list([Head|Tail]):-
            nested_list(Tail).
    sublist([Head|Tail]).
```

b) Test All Elements

```
    all_ok([]).
    all_ok([Head|Tail]):-
            one_ok(Head),
            all_ok(Tail).
```

plus definition of one_ok/1.

c) Return a Result -Having Processed One Element

```
    everything_after_a([a|Result],Result).
    everything_after_a([Head|Tail],Ans):-
            everything_after_a(Tail,Ans).
```

d) Return a Result -Having Processed All Elements

```
    triple([],[]).
    triple([H1|T1],[H2,T2]):-
            process(H1,H2),
            triple(T1,T2).
```

where process/1 takes H1 as input and outputs H2

5.2.2 Calling Patterns

For any given predicate with arity greater than 0, each argument may be intended to have one of three calling patterns:

Input
Output
Indeterminate

5.2.3 Arithmetic

To implement a successor relation,

```
    successor(X,Y):-
            Y is X + 1.
```

where it is intended that successor/2 takes the first argument as input and outputs the second argument which is to be the next largest integer.

In the above, note that X + 1 is evaluated.

This means that you must use the stated calling pattern as to try to solve the goal "successor(X,7)" will lead to trying to evaluate X + 1 with X unbound.

Therefore is/2 must always be called with its second argument as an arithmetic expression which has any variables already bound.

5.2.4 Reconstructing Lists

a) Building Structure on the "Way Back"

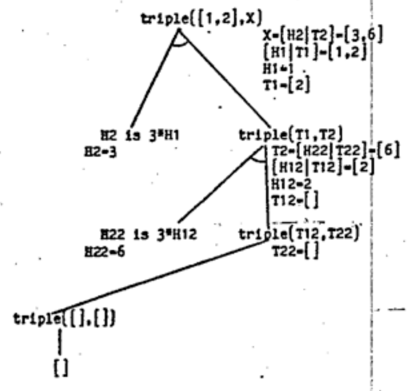
```
    triple([],[]).
    triple([H1|T1],[H2,T2]):-
            H2 is 3*H1,
            triple(T1;T2).
```

b) Building Structure on the "Way Down"

```
    reverse([],Y,Y).
    reverse([H|T],X,Y):-
            reverse(T,[H|X],Y).
```

This is left as an example for you to try to understand.

5.3 A Proof Tree



5.4 Conversations

```
    talk:-
            read_in(Sentence),
            process(Sentence,Reply),
            write_out(Reply),
            talk.
```

a) read_in/1

Converts characters read in to a list of Prolog atoms. Therefore "i hope you are well." turns into "[i,hope,you,are,well,.]". Note the "." as an atom.

b) write_out/1

Uses write/1 to write out the Reply -which is a list. So "[i,am,fine,.]" appears as "i am fine.".

c) process/2

Does some compulsory tidying using swap/2 and then uses a generator pick_up_transform/2 to produce a possible transformation rule which enables the modified input to be matched against its output. Note that any variables in the form of the output rule will only be bound after the match process.

```
     process(In,Out):-
          swap(In,ModifiedIn),
          pick_up_transform(PossibleRule,Out),
          match(PossibleRule,ModifiedIn).
```

d) swap/2

Some simple alterations.

```
     swap([i],[you]).
     swap([you],[i]).
     swap([you,are],[i,am]).
```

e) pick up transform/2

Here is an example rule.

```
     pick_up_transform([you,are,X,.],    [how,long,have,you,been,X,?]).
     pick_up_transform([X,is,Y.],        [is,X,always,Y,?]).
```

f) match/2

If the modified input were "[you,are,silly]" this would match the first argument of the first transformation rule "[you,are,X]" , X ought to be bound to "silly" and then the output half of the rule should produce "[how,long,have,you,been,silly,?]". An example matcher might be:

```
     match(X,X).
```

but it cannot cope with "[you,are,very,silly]" as this matcher can only match a logical variable with a single element of the list. A better one:

```
     match([Head|Rest],Fragment):-
          append(Head,Leftover,Fragment),
          match(Rest,Leftover).
     match([Head|Rest],[Head|Leftover]):-
          match(Rest,Leftover).
     match([],[]).
```

g) Finally

Just a mote about some low level I/O predicates:

[illegible]0(X)	unifies X with next non blank printable character (in ASCII code) from current input stream
[illegible])	unifies X with next character (in ASCII) from current input stream
[illegible]X)	puts a character on to the current output stream. X must be bound to a legal ASCII code

6 More Control and the Problem of Negation

6.1 Some Useful Predicates for Control

a) true/0

Always succeeds.

```
     father(jim,fred).
```

is logically equivalent to

```
     father(jim,fred):-
          true.
```

That is, any unit clause is equivalent to a non unit clause with a single subgoal "true" in the body.

b) fail/0

Always fails.

```
     lives_forever(X):-
          fail.
```

is intended to mean that any attempt to find an object X that lives forever will fail.

c) repeat/0

If it is asked to REDO then it will keep on succeeding.

```
     test:-
          repeat,
          write(hello),
          fail.
```

The goal "test" produces the output:

```
     hellohellohellohellohellohellohellohellohello...
```

repeat/0 behaves as if it were defined in Prolog as:

```
     repeat.
     repeat:-
          repeat.
```

d) call/1

The goal "call(X)" will call the interpreter as if the system were given the goal "X". Therefore X must be bound to a legal Prolog goal.

```
     ?- call(write(hello)).

     hello
     yes
```

6.2 The Problem of Negation

Consider

```
     man(jim).
     man(fred).

     ?- man(bert).

     no
```

[illegible]sk Prolog to solve a goal for which there is no clause then we assume [illegible] have provided Prolog with all the necessary data to solve the problem. [illegible] the Closed World Assumption.

[illegible] not/1

This takes a Prolog goal as its argument.

```
     ?- not( man(jim) ).
```

will succeed if "man(jim)" fails and will fail if "man(jim)" succeeds.

6.2.2 Negation as Failure

```
     man(jim).
     man(fred).
     woman(X):-
          not( man(X) ).

     ?- woman(jim).

     no
```

To solve the goal "woman(jim)" try solving "man(jim)". This succeeds therefore "woman(jim)" fails. Similarly, "woman(jane)" succeeds. But there is a problem. Consider:

```
     ?- woman(X).
```

It succeeds if "man(X)" fails -but "man(X)" succeeds with X bound to jim. So "woman(X)" fails and, because it fails, X cannot be bound to anything.

We can read "?- woman(X)" as a query "is there a woman?". Yet we know that "woman(jane)" succeeds. Therefore, this form of negation is not at all like logical negation if the argument of not/1 is a goal with an unbound variable. See section 6.4 for an extra comment on this.

Also, "not(not(man(X)))" is not identical to "man(X)" since the former will succeed with X unbound while the latter will succeed with X bound, in the first instance, to jim.

6.2.3 Negation as a Form of Case Selection

```
     parity(X):-
          odd(X),
          write(odd).
     parity(X):-
          not(odd(X)),
```

```
          write(even).

     plus set of facts defining odd/1
```

Provides extra expressivity as we do not need a set of facts to define even/1.

6.3 Some Abstract Program Schemata

a) Test - Process

```
     happy(X):-
          test1(X),
          process1(X).
     happy(X):-
          test2(X),
          process2(X).
```

Contrast with

b) Test - Generate

```
     select(X):-
          generate(X),
          test(X).
```

c) Commit

```
     find(X):-
          test(X),
          commit,
          process(X).

     plus clauses for find/1 etc
```

d) Satisfy Once Only

Sometimes, we would like a way of stopping Prolog looking for other solutions. Thart is, we want some predicate to be determinate.

```
     memberchk(X,[X|Y]):-
          make_determinate.
     memberchk(X<[Y|Z]):-
          memberchk(X,Z).
```

e) Abandon Hope

If we get to a position where we are certain that we want to give up trying to find a solution:

```
     attempt_solution(X):-
          happy(X),
          disaster(X),
          abandon_attempt.
```

We need ways of

COMMITting

Making DETERMINATE

ABANDONning a Goal

6.4 More on Negation

The negation implemented in Prolog

```
not(man(X))
```

usually has the following semantics: "it is not the case that there exists an object which is a man" which is equivalent to "for every object, it is not the case that it is a man". That is, the goal "not(man(X))" succeeds if there no known objects that were men ¬therefore no clauses for man/1.

7 Parsing in Prolog

Later on in the course, you will be involved in trying to face up to the problem of parsing ordinary english language sentences. For this lecture, we shall also be interested in parsing sentences but we will look at the very simplest examples.

First, what do we want the parser to do? We would like to know that a sentence is correct according to the (recognised) laws of english grammar.

The ball runs fast

is syntactically correct while

The man goes pub

is not as the verb "go" does (usually) not take a direct object.

Secondly, we may want to build up some structure which describes the sentence ¬so it would be worth returning, as a result of the parse, an expression which represents the syntactic structure of the successfully parsed sentence.

Of course, we are not going to try to extract the meaning of the sentence so we will not consider attempting to build any semantic structures.

7.1 Simple English Syntax

The components of this simple syntax will be such categories as sentences, nouns, verbs etc. Here is a (top down) description:

Unit: sentence
Constructed from: noun phrase followed by a verb phrase

Unit: noun phrase
Constructed from: proper noun or
determiner followed by a noun

Unit: verb phrase
Constructed from: verb or
verb followed by noun phrase

Unit: determiner
Examples: a, the

Unit: noun
Examples: man, cake

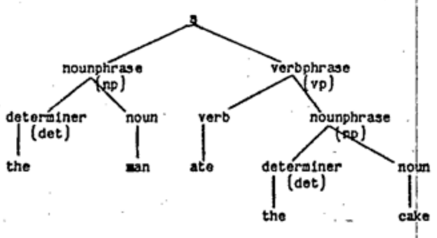
Unit verb:
Examples: ate

7.2 The Parse Tree

Here is a tree for

s = the man ate the cake

with some common abbreviations in brackets.



7.3 First Attempt at Parsing

We assume that we will parse sentences converted to list format.

We use append/3 to glue two lists together. The idea is that append returns the result of glueing takes input as lists in the first and second argument positions and returns the result in the third position.

```
sentence(S):-
        append(NP,VP,S),
        noun_phrase(NP),
        verb_phrase(VP).

noun_phrase(NP):-
        append(Det,Noun,NP),
        determiner(Det),
        noun(Noun).

verb_phrase(VP):-
        append(Verb,NP,VP),
        verb(Verb),
        noun_phrase(NP).

determiner([a]).
determiner([the]).
noun([man]).
noun([cake]).
verb([ate]).
```

Here is what happens to the query:

```
?- sentence([the,man,ate,the cake]).
```

```
append/3 succeeds with NP=[], VP=[the,man,ate,the,cake]
noun_phrase/1 fails
append/3 succeeds with NP=[the], VP=[man,ate,the,cake]
noun_phrase/1 fails
append/3 succeeds with NP=[the,man], VP=[ate,the,cake]
noun_phrase/1 succeeds
verb_phrase/1 succeeds
```

This is all very well but the process of parsing with this method is heavily non deterministic.

Also, it suffers from not being a very flexible way of expressing some situations. For example, the problem of adjectives:

, the quick fox

is also a noun phrase.

We might try to parse this kind of noun phrase with the extra clause:

```
noun_phrase(NP):-
        append(Det,Bit,NP),
        determiner(Det),
        append(Adj,Noun,Bit),
        adjective(Adj),
        noun(Noun).
```

A little ungainly.

7.4 A Second Approach

We now try an approach which is less non-deterministic. We will start by looking at:

```
sentence(In,Out)
```

The idea is that sentence/2 takes in a list of words as input, finds a legal sentence and returns a result consisting of the input list minus all the words that formed the legal sentence.

We can define it:

```
sentence(S,S0):-
        noun_phrase(S,S1),
        verb_phrase(S1,S0).
```

Here is a rough semantics for sentence/2.

> A sentence can be found at the front of a list of words
> if there is a noun phrase at the front of the list and
> a verb phrase immediately following.

This declarative reading should help to bridge the gap between what we want to be a sentence and the procedure for finding a sentence.

Here is the rest of the parser:

```
noun_phrase(NP,NP0):-
        determiner(NP,NP1),
        noun(NP1,NP0).

   hrase(VP,VP0):-
        verb(VP,VP1),
        noun_phrase(VP1,VP0).

determiner([a|Rest],Rest).
determiner([the|Rest],Rest).
noun([man|Rest],Rest).
noun([cake|Rest],Rest).
verb([ate|Rest],Rest).
```

As you can see, there is a remarkable sameness about the rules which, once you see what is going on, is fairly tedious to type in every time. So we turn to a facility that is built in to Prolog:

7.5 Prolog Grammar Rules

Prolog, as a convenience, will do most of the tedious work for you. What follows, is the way you can take advantage of Prolog.

This is how we can define the simple grammar:

```
sentence     -->  noun_phrase, verb_phrase.        = -->
noun_phrase  -->  determiner, noun.
verb_phrase  -->  verb, noun_phrase.
determiner.  -->  [a].
determiner   -->  [the].
noun         -->  [man].
noun         -->  [cake].
verb         -->  [ate].
```

It is very easy to extend if we want to include adjectives.

```
noun_phrase  -->  determiner, adjectives, noun.
adjectives   -->  adjective.
adjectives   -->  adjective, adjectives.
adjective    -->  [young].
```

We might later think about the orfdering of these rules and whether they really capture the way we use adjectives in general conversation but not now.

Essentially, the Prolog Grammar Rule formulation is <u>syntactic sugaring</u>. This means that Prolog enables you to write in:

```
sentence     -->  noun_phrase, verb_phrase.
```

and Prolog turns this into:

```
sentence(S,S0):-
        noun_phrase(S,S1),
        verb_phrase(S1,S0).
```

and

```
adjective    -->  [young].
```

into

```
adjective(A,A0):-
        'C'(A,young,A0).
```

where 'C'/3 is a built in Prolog Predicate which is defined as if:

```
'C'([H|T],H,T).
```

7.6 To Use the Grammar Rules

Set a goal of the form

```
sentence([the,man,ate,a,cake],[])
```

and not as

```
sentence.
```

or

```
sentence([the,man,ate,a,cake])
```

7.7 How to Extract a Parse Tree

We can add an extra argument which can be used to return a result.

```
sentence([[np,NP],[vp,VP]])            -->
        noun_phrase(NP),
        verb_phrase(VP).
noun_phrase([[det,Det],[noun,Noun]]) -->
        determiner(Det),
        noun(Noun).
   rminer(the)                         -->
        [the].
  d so on
```

What we have done above is declare predicates sentence/3, noun_phrase/3, verb_phrase/3, determiner/3 and so on. The explicit argument is the first and the two others are added when the clause is read in by Prolog. Basically, Prolog expands a grammar rule with n arguments into a corresponding clause with n+2 arguments.

So what structure is returned from

```
sentence(Structure,[the,man,ate,a,cake],[]).
```

The result is:

```
[[np,[[det,the],[noun,man]]],[vp,[...
```

Not too easy to read!

7.8 Adding Arbitrary Prolog Goals

Grammar rules are simply expanded Prolog goals. We can insert arbitrary Prolog subgoals on the right hand side of a grammar rule but we must tell Prolog that we do not want them expanded. For example, here is a grammar rule which parses a single character input and succeeds if the character is a digit. It also returns the digit found.

```
digit(D)  -->
        [X],
        { X >= 48,
          X =< 57,
          D is X-48 }.
```

The grammar rule looks for a character at the head of a list of input characters and succeeds if the Prolog subgoals

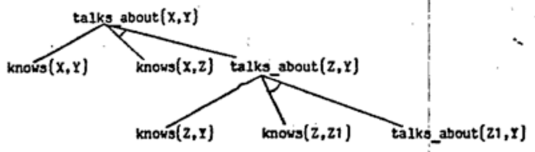
```
        { X >= 48,
          X =< 57,
          D is X-48 }.
```

succeed. Note that we assume we are working with ASCII codes for the characters and that the ASCII code for "0" is 48 and for "9" is 57. Also note the strange way of signifying "equal to or less than" as "=<".

8 Trees and Extralogical Operations

8.1 Search Space

```
talks_about(X,Y):-
        knows(X,Y).
talks_about(X,Y):-
        knows(X,Z),
        talks_about(Z,Y).
```



```
knows(a,b).
knows(b,c).
knows(b,d).
```

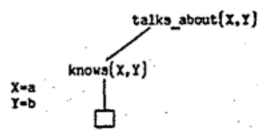
a) For a given tree, the tree has no existence <u>prior</u> to the execution.

b) We have only marked in choices with <u>possibly</u> matching heads.

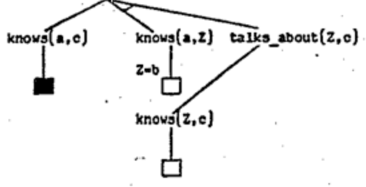
c) We must make it clearer as to what facts are available as part of the search space.

8.2 The Proof Tree

a) For Goal "talks_about(X,Y)"



b) For Goal "talks_about(a,c)"



a) We have propagated all the top level bindings. This is fair enough.

b) See that the Proof Tree does not carry the same information as the Search Tree.

c) Strictly, we should not show the failed OR choice.

8.3 Some Extralogical Control Predicates

We are looking for the solutions to problems posed in lecture 6.

```
     commit
     make_determinate
     abandon_attempt
```

8.3.1 Commit

Assume we want to make Social Security payments. That is, "pay(X,Y)" means "pay the sum X to Y".

```
     pay(X,Y):-
             british(X),
             entitled(X,Details,Y).
     pay(X,Y):-
             european(X),
             entitled(X,Details,Y).
```

If you check a person who is British and, for some reason, the subgoal "entitled(X,DFetails,Y)" fails then there is no point in checking if they are "european" (assuming that the sets of british and europeans are disjoint). We want to be committed to the OR choice for the pay/2 predicate.

The solution uses ! (cut).

```
     pay(X,Y):-
             british(X),
             !,
             entitled(X,Details,Y).
     pay(X,Y):-
             european(X),
             !,
             entitled(X,Details,Y).
```

8.3.2 Make Determinate

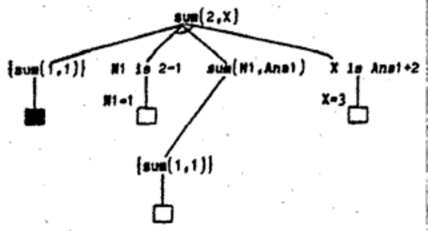
Consider:

```
     sum(1,1).
     sum(N,Ans):-
             N1 is N-1,
             sum(N1,Ans1),
             Ans is Ans1+N.
```

and the goal

```
     ?- sum(2,X).
```

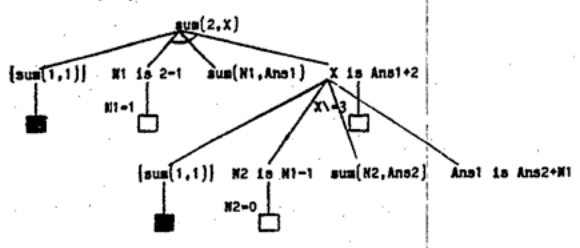
Here is the proof tree:



The goals in brackets should not strictly be there. They are for explanation only.

Now look at the goal:

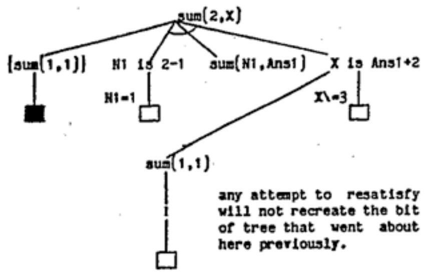
```
     ?- sum(2,X),fail.
```



Prolog goes into a non terminating computation. We want to make sure that, having found a solution, Prolog never looks for another solution via REDOing the goal. Here is the solution.

```
     sum(1,1).
             !.
     sum(N,Ans):-
             N1 is N-1,
             sum(N1,Ans1),
             Ans is Ans1+N.
```

The new proof tree:



sum(2,X)
{sum(1,1)}
N1 is 2-1
sum(N1,Ans1)
X is Ans1+2
N1=1
X\=3
sum(1,1)
!
any attempt to resatisfy will not recreate the bit of tree that went about here previously.

8.3.3 Abandon Attempt

Here is a way of defining woman/1 in terms of man/1.

```
     woman(X):-
             man(X),
             !,
             fail.
     woman(X).
```

To solve for "woman(jim)" we try "man(jim)". If that succeeds then we want to abandon the attempt to prove "woman(jim)" without trying any other clauses for woman/1.

We call this the cut-fail technique.

The above is a special case of Negation as Failure. Here is a possible definition of not/1 using cut (!) and call/1.

```
     not(Goal):-
             call(Goal),
             !,
             fail.
     not(Goal).
```

9 Prolog Syntax

Prolog Terms are one of:

```
     Constant
     Variable
     Compound Term
```

9.1 Constants

A Constant is one of:

```
     Atom
     Integer
     Real Number
```

Atoms are made up of:

letters and digits	AB...Zab...z01...9
signs	+-*/<>^:.?@$&
quoted strings	'any old character'

Normally, atoms start with a lower case letter. Note that, in a quoted atom, you can include a "'" by prefixing it with another "'". So, to print a "'" on the screen you will need a goal like "write('''')".

9.2 Variables

Variables usually start with a capital letter. The only interesting exception is the special anonymous variable written "_" and pronounced "underscore". In the rule

```
     process(X,Y):-
             generate(_,Z),
             test(_,Z),
             evaluate(Z,Y).
```

the underscores refer to different unnamed variables. For example, here are two versions of member/2.

```
     member(X,[X|Y]).
     member(X,[Y|Z]):-
             member(X,Z).

     member(X,[X|_]).
     member(X,[_|Z]):-
             member(X,Z).
```

Note that, in the clause,

```
     know_both_parents(X):-
             mother(_,X),
             father(_,X).
```

the underscores do not refer to the same object. The reading is roughly that "we know both the parents of X if someone(name unimportant) is the mother of X and someone else is the father".

9.3 Compound Terms

A Compound Term is a functor with a (fixed) number of arguments each of which may be a Prolog Term.

"This means that we can arbitrarily nest compound terms.

For some examples:

```
happy(fred)
              principal functor = happy
              1st argument      = a constant (atom)

sum(5,X)
              principal functor = sum
              1st argument      = constant (integer)
              2nd argument      = variable

not(happy(woman))
              principal functor = not
              1st argument      = compound term
```

Nesting compound terms may be of use to the programmer. For example, the clause

```
fact(fred,10000).
```

is not as informative as

```
fact(name(fred),salary(10000)).
```

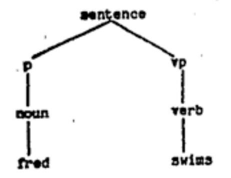
which can be thought of as defining a PASCAL-type record structure.

9.4 (Compound) Terms as Trees

Take the compound term

```
sentence(np(noun(fred)),vp(verb(swims)))
```

and construct a tree. Start by marking the root of the tree with the principal functor and draw as many arcs as the principle functor has arguments. For each of the arguments, repeat the above procedure.



9.5 Compound Terms and Unification

Consider

```
?- happy(X)=sad(jim).
```

-fails, because we know that it is necessary that the principal functors and their arities are the same for unification to succeed.

```
?- data(X,salary(10000))=data(name(fred),Y).
```

-succeeds, because, having matched the principal functors (and checked that the arities are the same) we recursively try to match corresponding arguments. This generates two subgoals:

```
X = name(fred)
salary(10000) = Y
```

which both succeed.

9.6 The Occurs Check

This is an aside. If we try to unify two expressions we must generally avoid situations where the unification process tries to build infinite structures. Consider:

```
data(X,name(X)).
```

and try:

```
?- data(Y,Y).
```

First we successfully match the first arguments and Y is bound to X. Now we try to match Y with name(X). This involves trying to unify name(X) with X. What happens is an attempt to identify X with name(X) which yields a new problem -to match name(X) against name(name(X)) and so on.

To avoid this it is necessary, that, whenever an attempt is made to unify a variable with a compound term, we check to see if the variable is contained within the structure of the compound term.

Most prolog implementations have deliberately missed out the occurs check -mostly because it is computationally very expensive.

9.7 Lists Are Terms Too

If a list is a term then it must be a compound term. What, then is its principal functor? Lists can be any length -so what is the arity of the principle functor.

For the moment only, let us suppose we have a gluing agent which glues an element onto the front of a list. We know this is a reasonable supposition because we already have a list destructor/constructor that works like this.

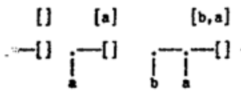
```
[a,b,c,d] = [Head|Tail]
-results in Head=[a], Tail=[b,c,d]
```

Think of this constructor as a predicate cons/2. We have to build lists like this.

The empty list is referred to as [] or, sometimes, the atom "nil" -but the Prolog you will use does not refer to the empty list as nil.

Familiar List Notation	Intermediate Form	Compound Term Form
[ ]		nil
[a]		cons(a,nil)
[b,a]	cons(b,[a])	cons(b,cons(a,nil))
[c,b,a]	cons(c,[b,a])	cons(c,cons(b,cons(a,nil)))

Now to represent the lists as trees -but we will distort them a little:

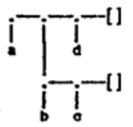


You will have noticed that I should have written "cons" where I have written ".". Well, the truth is, Prolog doesn't use a meaningful name for the constructor cons/2. Really, the constructor is '.'/2. For explanation purposes, I shall stick to using cons/2.

Now for a non-flat list

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

cons(a,[[b,c],d])

cons(a,cons([b,c],[d])
        now [b,c] is cons(b,[c])
        that is,     cons(b,cons(c,nil))
cons(a,cons(cons(b,cons(c,nil)),[d])

cons(a,cons(cons(b,cons(c,nil)),cons(d,nil)))
```

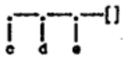
Now construct the tree using the method for drawing trees of compound terms.

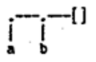


9.8 How To Glue Two Lists Together

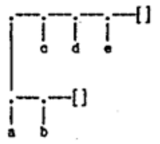
We want to "glue", say, [a,b] to [c,d,e] to give the result [a,b,c,d,e]. That is, we want a predicate append/3 taking two lists as input and returning the third argument as the required result.

Here are the two lists as trees:

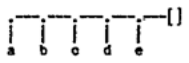




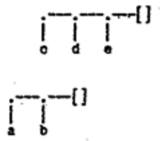
You might think of checking to see whether "cons([a,b],[c,d,e])" correctly represents the list "[a,b,c,d,e]". Look at this solution as a tree.



It is not the required



Let's try again:



We could solve our problem in a procedural manner using our list deconstructor as follows:

```
Lop off the head "a" of the first list "[a,b]"
  Solve the subproblem of gluing "[b]" to "[c,d,e]"
Put the head "a" back at the front of the result

But we have a subproblem to solve:

Lop off the head "b" of the first list "[b]"
  Solve the subproblem of gluing "[]" to "[c,d,e]"
Put the head "a" back at the front of the result

But we have a subproblem to solve:
Gluing "[]" to "[c,d,e]" is easy..the result is "[c,d,e]"
```

First thing to note is that there is a recursive process going on. It can be read as:

Take the head off the first list and keep it until we have solved the subproblem of gluing the rest of the first list to the second list. To solve the subproblem simply apply the same method.

Once we are reduced to adding the empty list to the second list, return the solution -which is the second list. Now, as the recursion unwinds, the lopped off heads are stuck back on in the correct order.

Here is the code:

```
     append([],List2,List2).
     append([Head|List1],List2,[Head|List3]):-
             append(List1,List2,List3).
```

9.9 Rules as Terms

Consider:

```
     happy(X):-
             rich(X).
```

If this is a term then it is a compound term. Again, what is its principal functor and its arity?

1. Principal Functor is

```
     :-
```

Usually, the functor is written in infix form rather than the more usual prefix form.

2. Arity is

```
     2
```

3. The above rule in prefix form

```
     ':-'(happy(X),rich(X)).
```

But what about

```
     happy(X):-
             healthy(X),
             wealthy(X),
             wise(X).
```

Trying to rewrite in prefix form:

```
     ':-'(happy(X),whatgoeshere?).
```

Note that the comma ',' in this expression is an argument separator. In the definition of happy/1 above, the commas are read as "and".

Yes,

```
     healthy(X),wealthy(X),wise(X).
```

is also a compound term with principal functor

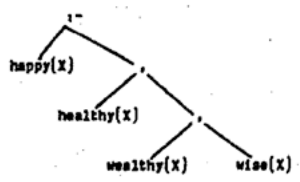
```
     ,
```

and arity 2. Since we have to represent three subgoals and the arity of ',' is 2 we again have a nested compound term. The correct prefix form for the example is:

```
     ','(healthy(X),','(wealthy(X),wise(X))).
```

Note: try the goal "display((healthy(X),wealthy(X),wise(X)))" to see the "truth". Also, note that, for a reason as yet unexplained, you need an extra pair of brackets around the goal you want printed via display/1.

Here is the tree:



10 Input/Output

We will are going to discuss a number of practical issues.

10.1 Testing a Predicate

Suppose that we want to test the predicate double/2.

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

To do this, we write a test predicate:

```
     test:-
             read(X),
             double(X,Y),
             write(Y),
             nl.
```

Here is a transcription of a "test".

```
     ?- test.
     |: 2.
     4
     yes
```

Note that, since we are using read/1 which only accepts valid Prolog terms terminated by a "." followed by <RETURN> (in this case), we have to enter such as "2.".

Now to add a loop. The easy way is to recursively call test/0. We would prefer, however, to put in a test so that we can abort the loop.

```
     test:-
             read(X),
             X \= -1,
             double(X,Y),
             write(Y),
             nl,
             test.
```

The predicate \=/2 is written in infix form and succeeds only if it is impossible to unify the two arguments (which may legitimately be any two Prolog terms). Thus "X \= Y", "2 \= X" and "X \= 2" all fail while "fred \= jim" succeeds.

When we quit the test above the goal "test" will fail since there are no choices to remake.

10.2 Input/ Output Channels

The Standard Input stream is taken from the keyboard and is known as "user".

Think of the stream of characters typed in as issuing from a file called "user".

The Standard Output stream is directed to the terminal screen and is known as "user" too.

Think of the stream of characters issuing from Prolog as going to a file called "user".

10.3 Input/ Output and Files

Let us take our input data from a file called "in".

```
     go:-
             see(in),
             test,
             seen.
```

We "wrap" the test/0 predicate into a predicate go/0 which takes input fr the specified file "in". This file should contain legal Prolog terms -for t predicate double/2 we want something like:

```
     2.
     23.
     -1.
```

see/1	Take input from the named file
seen/0	Close the current input stream and take input from user

How do you find out what the current input stream is?

seeing/1	Returns name of current input stream

Now to redirect output to a file named "out":

```
     go:-
             tell(out),
             see(in),
             test,
             seen,
             told.
```

Using the same file "in" as previously, "out" will contain:

```
     4
     46
```

tell/1	Send output to the named file
told/0	Close the current output stream and send output to user

How do you find out what the current output stream is?

telling/1	Returns name of current output stream

10.4 The End of File Marker

When read/1 encounters the end of a file it returns the Prolog atom

```
     end_of_file
```

So we can rewrite test/0:

```
    test:-
        read(X),
        X \= end_of_file,
        double(X,Y),
        write(Y),
        nl,
        test.
```

10.5 Input of Prolog Terms

Both consult/1 and reconsult/1 have been described in a handout. Prolog will try to read a clause at a time from the named file. So any error message only refers to the current term being parsed.

Of course, if Prolog cannot find the end properly then we have problems. The Prolog you are using will load all clauses that parse as correct and throw away any ones that do not parse.

Some example problems:

a)

```
a:-                         a:-
        b,                          b,
        c,      is read as          c,
d:-                                 d:-e.
        e.
```

There are problems with this reading which will be reported by Prolog

b)

```
a:-                         a:-
        b.                          b.
        c,      is read as  c,d:-e.
d:-
        e.
```

This is basically illegal.

10.6 Some Example Programs

10.6.1 The Predicate remove/3

[illegible]ove all instances of a named element from a list and return the [illegible]er as a list.

[illegible] need to use recursion.

If so, we will need to recurse down the input list, processing an element at a time. Basically, the output is a rough copy of the input.

A typical goal will be:

```
    remove(fred,[fred,jimmy,fred,bill],X)
```

Case 1

```
    remove(Element,[],[]).
```

the empty list is the result of removing "Element" from the empty list.

Case 2

```
    remove(Element,[Element|Rest],Ans):-
        remove(Element,Rest,Ans).
```

The result of removing the element from the whole list is the same as the result of removing the element from the beheaded list if the element is the head of the list.

Case 3

```
    remove(Element,[X|Rest],[X|Ans]):-
        remove(Element,Rest,Ans).
```

Otherwise, the result is the head of the input list appended to the result of removing the element from the beheaded list.

We found the answer by building "bits" of the wanted result and going off to look for the rest.

We can also solve the same problem by carrying the partial answer along with the various subgoals until the whole answer has been found. We will have to carry around a fourth argument which will be uninstantiated until all the answer has been gathered.

10.6.2 The Predicate remove/3

First, we define the predicate remove/4 A typical goal will be:

```
    remove(fred,[fred,jimmy,fred,bill],[],X)
```

We can package it to be like remove/3 by:

```
    remove(El,Inlist,Outlist):-
        remove(El,Inlist,[],Outlist).
```

Case 1

```
    remove(Element,[],Ans,Ans).
```

the answer is the accumulator if we have no more list to process.

Case 2

```
    remove(Element,[Element|Rest],Acc,Ans):-
        remove(Element,Rest,Acc,Ans).
```

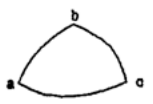
the same as before

Case 3

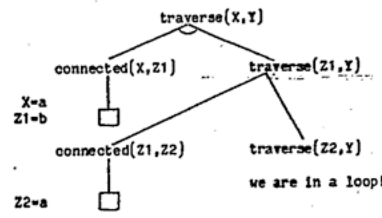
```
    remove(Element,[X|Rest],Acc,Ans):-
        remove(Element,Rest,[X|Acc],Ans).
```

Note how we say that the accumulator has the head of the input added to it. In this version of "remove", structure is being built up in the body of the clause rather than the other version where the structure is built in the head.

10.7 Traversing a Graph



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

    traverse(X,Y):-
        connected(X,Z),
        traverse(Z,Y).
```



So we add an extra argument to store information about where we have been and try to make sure that we never revisit that place again.

```
    traverse(X,Y,Placesvisited):-
        connected(X,Z),
        not( member(Z,Placesvisited)),
        traverse(Z,Y,[Z|Placesvisited]).
```

where member/2 has been met before and we have the same connectivity facts. We can call it with

```
    traverse(X,Y,[]).
```

It does not work. Eventually, it will not be able to find a place to go that it has not been. Prolog will backtrack and fail to find an answer. As an exercise, you can write a version that will work!

11 Operators

An operator is a predicate which has some special properties.

Here is a list of ones we have met already:

```
        +   -   *   /

        =           \=

        <   =<  >   >=

            is

        (not)       \+

            ,

            -->

            :-

            ?-
```

Note that not/1 is in brackets because the Prolog you will be using does not have not/1 built in.

11.1 The Three Forms

11.1.1 Infix

Here are some examples:

```
    3 + 2       23 - 2      8 * 2       30 / 2

    2 < 7       6 > 2       Y is 23

    healthy(jim),wealthy(fred)      adjective --> [clever]

    a:-b
```

11.1.2 Prefix

```
    \+ man(jane)
    not happy(fred)
```

11.1.3 Postfix

We have not seen this one -but it might have existed!

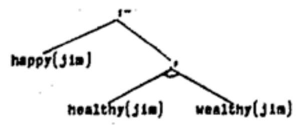
```
    n !  (factorial)
```

11.2 Precedence

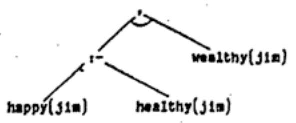
We will now look at the structure of some Prolog expressions:

```
    happy(jim):-
        healthy(jim),
        wealthy(jim).
```

We assume that it is always possible to represent a Prolog expression as a tree in an unambiguous way. Is this



or



The issue is decided by <u>operator</u> <u>precedence</u>.

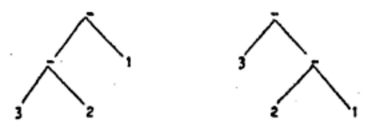
To construct a tree which describes a Prolog expression we first look for the operator with the highest precedence. If this operator is an infix one, we can divide the expression into a left hand one and a right hand one. The process is then repeated, generating left and right subtrees.

operator	precedence
-->	1200
:-	1200
,	1000
\+	900
is	700
<	700
=	700
=<	700
>	700
>=	700
=	700
+	500
-	500
*	400
/	400

We still need to decide what to do with two operators of the same precedence. Should we regard

```
    3 - 2 - 1
```

as or



and, remember, that we are not yet talking about arithmetic evaluation!

We can use brackets to distinguish

```
    (3 - 2) -1      from     3 - (2 - 1)
```

but we have a special way of distinguishing which interpretation we wish Prolog to make. In the above arithmetic example, the left hand tree has two subtrees hanging from the root "-". The left hand one has "-" as its root while the right hand one is not so allowed. We say that this interpretation of "-" is <u>left</u> <u>associative</u>.

The normal interpretation of "-" is left associative. The common left associative operators are:

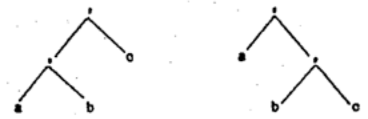
```
    *
    /
    +
    -
    div      (integer division)
```

Are there any <u>right</u> <u>associative</u> operators? Yes —consider how we are to disambiguate

```
    a,b,c
```

where "a", "b" and "c" are all legal Prolog subgoals.

is it or



ie (a,b),c or a,(b,c)
(left associative) (right associative)

The answer is that ,/2 is right associative.

In all the previous cases we have allowed exactly one subtree to have, as its root, the same operator as the "principal" root. We can extend this to permit operators of the same precedence. Thus, since "+" and "-" have the same precedence, we know that

```
    3 - 2 + 1
```

is left associative (and legal) and represents (3 - 2) +1.

Sometimes, we do not wish to permit left or right associativity. For example, obvious interpretations of:

```
    a:- b :- c
    Y is Z+1 is 3
    a --> b --> c
```

do not readily spring to mind. Therefore we make it possible to forbid the building of expressions of this sort.

11.3 <u>Associativity</u> <u>Notation</u> <u>for</u> <u>Infix</u> <u>Operators</u>

Left Associative	yfx
Right Associative	xfy
Not Associative	xfx

Note that x indicates that the indicated subtree must have, as its root, an operator of lower precedence than that of the root.

The y indicates that the root of the subtree may have the same precedence as the operator that is the root of the tree.

The f indicates the operator itself.

11.4 The <u>Prefix</u> <u>Case</u>

Here are a number of unary, prefix operators:

operator	precedence
:-	1200
?-	1200
\+	900
not	900
unary +	500
unary -	500

We regard a prefix operator as having only a right hand subtree. We must decide which of the above may be right associative. That is, which of the following make sense:

```
    + + 1
    not not happy(jim)
    :- :- a
```

We only accept not/1 and \+/1 as right associative.

11.5 <u>Associativity</u> <u>Notation</u> <u>for</u> <u>Prefix</u> <u>Operators</u>

Right Associative	fy
Not Associative	fx

11.6 <u>Associativity</u> <u>Notation</u> <u>for</u> <u>Postfix</u> <u>Operators</u>

As we have no examples here at the moment, here is the table:

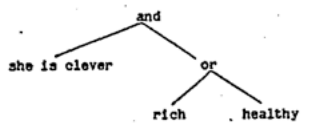
Right Associative	yf
Not Associative	xf

11.7 <u>How</u> <u>to</u> <u>Change</u> <u>Operator</u> <u>Definitions</u>

We will illustrate with an infix operator and/2 and another or/2. We will choose the precedence of and/2 to be greater than that of or/2. This means that we interpret:

```
    she is clever and rich or healthy
```

as



Since and/2 reminds us of ,/2 we will give it the same precedence and associativity:

```
    Precedence    of and/2 = 1000

    Associativity of and/2 = xfy
```

The required command is

```
    op(1000,xfy,and).
```

We could also make it like ,/2 by <u>interpreting</u> and/2 as in:

```
    X and Y :-
        call(X) , call(Y).
```

For or/2 we choose <u>precedence</u> of 950 (less than and/2) and <u>associativity</u> of xfy (the same as and/2) with:

```
    op(950,xfy,or)
```

and define it as equivalent to:

```
    X or Y :-
        call(X).
    X or Y :-
        call(Y).
```

11.8 <u>A</u> <u>More</u> <u>Complex</u> <u>Example</u>

We now try to represent data structures that look like:

```
    if a and b or c then d
```

[illegible] we already have a representation for "a and b or c", this reduces to representing

```
     if a then b
```

We will make "then" an infix operator of arity 2. Because both subtrees might contain and/2 we will need to make then/2 of higher precedence than and/2 -say,1050 and not associative. Hence:

```
     op(1050,xfx,then)
```

This means that "if" must be a prefix operator. As we do not wish expressions of the form

```
     if if a
```

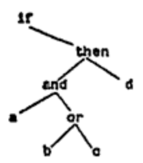
we must make if/1 of higher precedence than then/2 (say, 1075) and if/1 must be non associative:

```
     op(1075,fx,if)
```

We can now represent

```
     if a and b or c then d
```

as the tree



or, as the Prolog term

```
     if(then(and(a,or(b,c)),d))
```

This Prolog term is difficult to read but unambiguous while the representation using operators is easy to read but depends heavily on you understanding the precedences and associativities involved. All right if you wrote the code but th[illegible] is harder for someone else to read.

12 Conclusion: Where Next?

We discuss some powerful features that Prolog offers then the important subject of programming style. Finally, some aspects of Prolog are mentioned that demonstrate that the development of Logic Programming is by no means over.

12.1.1 Powerful Features -Typing

All these features are not strictly first order predicate logic. Nevertheless they give great power into the hands of the programmer.

predicate/arity	succeeds if the argument is
atom/1	atom
integer/1	integer
atomic/1	atom or integer
var/1	uninstantiated variable
nonvar/1	not an uninstantiated variable

12.1.2 Powerful Features -Splitting Up Clauses

a) clause/2

```
     happy(X):-
             healthy(X),
             wealthy(X).
     happy(jim).
```

The goal "clause(happy(X),Y)" produces

```
     Y = healthy(X), wealthy(X)
```

on redoing,

```
     Y = true
```

Note the second answer returns a body of "true" for the clause "happy(jim)".

The calling pattern requires that the principal functor of the first argument is known.

b) functor/3

```
     fact(male(fred),23).

     ?- functor(fact(male(fred),23),F,N).

     F=fact
     N = 2
```

functor/3 can be used to find the principal functor of a compound term together with its arity. It can also be used to generate structures:

```
     ?- functor(X,example,2).

     X = example(A,B)
```

except that the variables will be shown differently.

c) arg/3

```
     fact(male(fred),23).

     ?- arg(1,fact(male(fred),23),F).

     F = male(fred)
```

arg/3 is used to access a specified argument for some Prolog term.

d) =../2

[illegible]..." is pronounced "univ".

```
     [illegible] X=.. [fact,male(fred),23].

     [illegible] = fact(male(fred),23)

     ?- (a + b) =.. X.

     X = [+, a, b]

     ?- [a,b,c] =.. X.

     X = ['.',a,[b,c]]
```

12.1.3 Powerful Features -Comparisons of Terms

a) ==/2

If you do not want to unify two Prolog terms but you want to know if the terms are strictly identical.

```
     ?- X == Y.

     no

     ?- X=Y, X == Y.

     yes
```

b) \==/2

This is equivalent to the Prolog definition

```
     X \== Y:-
             \+ (X == Y).
```

12.1.4 Powerful Features -Finding All Solutions

a) setof/3

The semantics for setof/3 are unpleasant. It has to be used with care.

```
     knows(jim,fred).
     knows(alf,bert).
```

How do we find all the solutions of the form "knows(X,Y)"? Now the goal "knows(X,Y)" is equivalent to asking "does there exist some X and some Y such

that knows(X,Y)". For all solutions we want to ask something like "for what set of values of X and set of values of Y is it true that for all X and all Y then knows(X,Y)".

```
     setof([X,Y],knows(X,Y),Z).

     Z = [[jim,fred],[alf,bert]]
```

where Z is the set of all solution pairs [X,Y] such that knows(X,Y). Now suppose we only want to gather the first element of the pairs.

```
     ?- setof(X,Y^knows(X,Y),Z).

     Z = [jim, alf]
```

You have to read this as "find the set Z consisting of all values of X for which there exists a value Y for which knows(X,Y)". The "Y^" is interpreted as "there exists a Y" and is vital.

Note that any repeated solutions are removed.

b) bagof/3

The only difference between bagof/3 and setof/3 is that bagof leaves repeated solutions in the answer. Note that bagof/3 is much less expensive than setof/3.

12.1.5 Powerful Features -Generating Known Terms

a) current_atom/1

```
     ?- current_atom([]).

     yes
```

b) current_functor/2

```
     ?- current_functor(atom,atom(fred)).

     yes
```

c) current_predicate/2

```
     knows(fred).

     ?- current_predicate(knows,knows(fred)).

     yes
```

d) current_op/3

```
     ?- current_op(1200,xfx,(:-)).

     yes
```

All the above can be used to generate information as well!

<u>12.2 Prolog Style</u>

<u>12.2.1 Comments</u>

a) End of Line Comments

```
append([],A,A).                $ the base case
append([A|B],C,[A|D]) :-
        append(B,C,D).         $ recurse on the first argument
```

Everything on the line after "$" will be ignored by Prolog

b) Section Comments

```
/*
we now define append/3 so that
it can be used as a generator
*/
```

Everything between the "/* ... */" will be ignored by Prolog. Best to put this just before the code discussed. This is also useful for program development.

<u>12.2.2 Program Headers</u>

Just a recommendation -not at all compulsory- but you will need something like it for yourself.

```
$ Program: pract2.pl
$ Author:  ecmu26
$ Date:    27 October 1985
$ Purpose: 2nd AI2 Practical
```

<u>12.2.3 Side Effect Programming</u>

Avoid where possible. Most of the time it is possible to avoid the worst offences.

a) Modifying the Program at Runtime

Prolog permits this but it is generally bad programming style. For example, Prolog will not automatically undo these changes on backtracking.

b) Wanting to Remember Something

A subset of the previous case. It is better to consider carrying around the wanted information as an extra argument in all the relevant clauses.

<u>12.2.4 Some Other Pointers</u>

a) cut

Use cuts with <u>great care</u>.

b) IF ...THEN & IF ..THEN ...ELSE

Prolog does support control structures of this form. You may be comfortable with them but it is better, if more cumbersome, to avoid them. Here is how one might define Prolog's "if ...then ...else".

```
(a -> b ; c) :-
      call(a),
      !,
      call(b).
(a -> b ; c) :-
      call(c).
```

c) ;/2

The semantics of ;/2 are roughly equivalent to logical or. Best to avoid its use.

```
a:- b ; c.
```

is better written as:

```
a:- b.
a:- c.
```

<u>12.3 Prolog and Logic Programming</u>

<u>12.3.1 Prolog and Resolution</u>

There are many different Prologs but they are all based on a technique from theorem proving known as SLD Resolution.

SLD resolution can be guaranteed to be <u>complete</u> in that if a solution exists then it can be found using some search strategy.

SLD resolution can be guaranteed to be <u>sound</u> in that if an answer is obtained then it is a solution to the original problem for some search strategy.

It is a research goal to study Prolog implementations and check that their search strategy preserves the completeness and soundness of the underlying method of SLD resolution.

Note that cut affects completeness but not soundness.

Note also that there is no theoretical way of determining whether or not an attempt to solve a problem will terminate. If there is a solution then it can be shown that it can be found in a finite number of steps.

<u>12.3.2 Prolog and Parallelism</u>

Various people are working on strategies for parallel execution of Prolog.

This includes Clarke and Gregory at Imperial College, London where much work has been done in developing PARLOG.

Ehud Shapiro of the Weizmann Institute, Israel has produced Concurrent Prolog.

<u>12.3.3 Prolog and Execution Strategies</u>

John Lloyd and others have produced MUProlog at the University of Melbourne in an attempt, inter alia, to replace the standard Prolog left-right execution strategy for subgoals with a strategy which can reorder the execution sequence depending on which subgoals have enough information to proceed with their execution.

<u>12.3.4 Prolog and Functional Programming</u>

Many attempts are being made to combine Prolog with functional programming features.

<u>12.3.5 Other Logic Programming Languages</u>

Prolog is not a pure logic programming language. It may be the best we have but there is some interest in building better languages.

As Prolog is less expressive than first order predicate calculus, a fair amount of work is going on to produce systems that permit the user to exploit the expressivity of full first order predicate logic -and other logics too!

BUILT-IN PREDICATES

Edinburgh Prolog v1.3+ is not quite DEC-10 -but close enough for now.

abolish(F,N)	Abolish the interpreted procedure named F arity N.
abort	Abort execution of the current directive.
ancestors(L)	The ancestor list of the current clause is L.
arg(N,T,A)	The Nth argument of term T is A.
assert(C)	Assert clause C.
assert(C,R)	Assert clause C,reference R.
asserta(C)	Assert C as first clause.
asserta(C,R)	Assert C as first clause, reference R.
assertz(C)	Assert C as last clause.
assertz(C,R)	Assert C as last clause, reference R.
atom(T)	Term T is an atom.
atomic(T)	Term T is an atom or integer.
bagof(X,P,B)	The bag of instances of X such that P is provable is B.
break	Break at the next interpreted procedure call.
call(P)	Execute the interpreted procedure call P.
clause(P,Q)	There is an interpreted clause, head P,body Q.
clause(P,Q,R)	There is an interpreted clause, head P, body Q, ref R.
close(F)	Close file F.
compare(C,X,Y)	C is the result of comparing terms X and Y.
compile(F)	Compile the procedures in text file F.
consult(F)	Extend the program with clauses from file F.
current_atom(A)	One of the currently defined atoms is A.
current_functor(A,T)	A current functor is named A, most general term T.
current_predicate(A,P)	A current predicate is named A, most general goal P.
current_op(P,T,A)	Atom A is an operator type T precedence P.
debug	Switch on debugging.
debugging	Output debugging status information.
depth(D)	The current invocation depth is D.
display(T)	Display term T on the terminal.
erase(R)	Erase the clause or record, reference R.
expand_term(T,X)	Term T is a shorthand which expands to term X.
fail	Backtrack immediately.
fileerrors	Enable reporting of file errors.
functor(T,F,N)	The principal functor of term T has name F, arity N.
get(C)	The next non-blank character input is C.
get0(C)	The next character input is C.
halt	Halt Prolog, exit to the monitor.
instance(R,T)	A most general instance of the record reference R is T.
integer(T)	Term T is an integer.
Y is X	Y is the value of integer expression X.
keysort(L,S)	The list L sorted by key yields S.
leash(M)	Set leashing mode to M.
length(L,N)	The length of list L is N.
listing	List the current interpreted program.
listing(P)	List the interpreted procedure(s) specified by P.
maxdepth(D)	Limit invocation depth to D.
name(A,L)	The name of atom or integer A is string L.
nl	Output a new line.
nodebug	Switch off debugging.
nofileerrors	Disable reporting of file errors.
nonvar(T)	Term T is a non-variable.
nospy P	Remove spy-points from the procedure(s) specified by P.
numbervars(T,M,N)	Number the variables in term T from M to N-1.
op(P,T,A)	Make atom A an operator of type T precedence P.
phrase(P,L)	List L can be parsed as a phrase of type P.
print(T)	Portray or else write the term T.
prompt(A,B)	Change the prompt from A to B.
put(C)	The next character output is C.
read(T)	Read term T.
reconsult(F)	Update the program with procedures from file F.
recorda(K,T,R)	Make term T the first record under key K, reference R.
recorded(K,T,R)	Term T is recorded under key K, reference R.
recordz(K,T,R)	Make term T the last record under key K, reference R.
reinitialise	Initialisation -looks for 'prolog.bin' or 'prolog.ini'.
rename(F,G)	Rename file F to G.
repeat	Succeed repeatedly.
restore(S)	Restore the state saved in file S.
retract(C)	Erase the first interpreted clause of form C.
save(F)	Save the current state of Prolog in file F.
save(F,R)	As save(F) but R is 0 first time, 1 after a 'restore'.
see(F)	Make file F the current input stream.
seeing(F)	The current input stream is named F.
seen	Close the current input stream.
setof(X,P,S)	The set of instances of X such that P is provable is S.
shell(T)	Allows certain interactions with the operating system.
skip(C)	Skip input characters until after character C.
sort(L,S)	The list L sorted into order yields S.
spy P	Set spy-points on the procedure(s) specified by P.
statistics	Output various execution statistics.
statistics(K,V)	The execution statistic key K has value V.
subgoal_of(G)	An ancestor goal of the current clause is G.
tab(N)	Output N spaces.
tell(F)	Make file F the current output stream.
telling(F)	The current output stream is named F.
told	Close the current output stream.
trace	Switch on debugging and start tracing immediately.
true	Succeed.
ttyflush	Transmit all outstanding terminal output.
ttyget(C)	The next non-blank character from the terminal is C.
ttyget0(C)	The next character input from the terminal is C.
ttynl	Output a new line on the terminal.
ttyput(C)	The next character output to the terminal is C.
ttyskip(C)	Skip over terminal input until after character C.
ttytab(N)	Output N spaces to the terminal.
unknown(O,N)	Change action on unknown procedures from O to N.
var(T)	Term T is a variable.
version	Displays introductory/system identification messages.
version(A)	Adds the atom A to the list of introductory messages.
write(T)	Write the term T.
writeq(T)	Write the term T, quoting names where necessary.
!	Cut any choices taken in the current procedure.
\+ P	Goal P is not provable.
X^P	There exists an X such that P is provable.
X<Y	As integer values, X is less than Y.
X=<Y	As integer values, X is less than or equal to Y.
X>Y	As integer values, X is greater than Y.
X=Y	Terms X and Y are equal (i.e. unified).
X==Y	Terms X and Y are strictly identical.
X\==Y	Terms X and Y are not strictly identical.
X@<Y	Term X precedes term Y.
X@=<Y	Term X precedes or is identical to term Y.
X@>Y	Term X follows term Y.
X@>=Y	Term X follows or is identical to term Y.
[F|R]	Perform the consult/reconsult(s) on the listed files.