Facts, Rules, Queries

- Prolog programs are built out of facts and rules; users formulate queries to state the program goals.
- Facts, rules, and queries are composed of the basic language elements (terms) – atoms, variables, and structures.

Programming in Prolog

- declare facts describing explicit relationships between objects and properties objects might have (e.g. Mary likes pizza, grass has_colour green, Fido is_a_dog, Mizuki taught Paul Japanese)
- define rules defining implicit relationships between objects (e.g. the sister rule above) and/or rules defining implicit object properties (e.g. X is a parent if there is a Y such that Y is a child of X).

One then uses the system by:

- asking questions above relationships between objects, and/or about object properties (e.g. does Mary like pizza? is Joe a parent?)

Prolog Facts

- A fact is **a term** followed by **a period**: *speaks (bob, english).* The arguments are atoms.
- Facts are similar to headless Horn clauses. There are built from propositions that are assumed to be true.
- Facts can't include **variables** [it specify Properties of objects, *or* relationships between objects];
- "Dr Turing lectures in course 9020", is written in Prolog as:
  lectures(turing, 9020).

□ *Notice that:*

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

- lectures(turing, 9020). is also called a *predicate*

Rules in Prolog

- a rule consists of three parts: **Head, Neck (if condition) , Body and a period.**
- A rule is a term followed by: - and one or more terms, ending in a period:
  term :- $term_1$, $term_2$, ... $term_n$.

grandparent (X,Z) :- parent(X, Y), parent(Y, Z).

- Rules are headed Horn clauses.
- The operator (neck) " :- " in a rule means *"if"*

Fact/Rule Semantics parent(X, Y). mean "X is the parent of Y", or "Y is the parent of X"?

- The semantics of Prolog rules and facts are supplied by the programmer.
- The Prolog system assumes that the left hand side (LHS) of a rule is true only if all the terms on the RHS are true.
- Use of variables makes rules general (provides a kind of universal quantification.)

  female(X) :- mother(X).

  is equivalent to

  $\forall x(\text{mother}(x)) \supset \text{female}(x)$
- The commas (,)that separate terms on the RHS represent the 'and' operator, so all must be true; in other words, the terms represent a conjunction of conditions.
- To express disjunction (or), use additional rules:

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

  parent(X,Y) :- father(X, Y).
- **Or** … use a **semicolon (;)** to indicate disjunction:

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

Facts, Rules, & Goal Statements

- Goals (queries) are propositions to be proven true or false, based on facts that are given as part of the program
- Syntax for a goal statement:
  - headless Horn clause: dog(spot).
  - a conjunction of clauses: dog(spot), owner(Who, spot).
  - clauses with several parameters: father(X, Y).

Success and Failure

- A rule *succeeds* when there are instantiations (temporary assignments to variables) for which all right-hand terms are true.
- A rule *fails* if it doesn't succeed.
- Facts always succeed.

```
speaks(allen, russian). speaks(bob,
english). speaks(mary, russian).
speaks(mary, english).
talkswith(X, Y):- speaks(X, L),
speaks(Y, L), X \= Y.
```

- This program has four facts and one rule.

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

?- *lectures(codd, 9020).*
false.

- if answer is true., the query *succeeded*
- if answer is false., the query *failed.* Note: many early versions of Prolog, including early versions of SWI-Prolog, say No instead of false. See the article on negation in the Prolog dictionary to find out why No. is a more accurate description of what is happening.
- In the latest version of SWI Prolog, it no longer says "No." but says "false." instead.
- The use of lower case for codd is critical.
- Prolog is not being intelligent about this - it would not see a difference between this query and lectures(fred, 9020). or lectures(xyzzy, 9020). though a person inspecting the database can see that fred is a student, not a lecturer, and that xyzzy is neither student nor lecturer.

Variables

- Suppose we want to ask, "What course does Turing teach"?
- This could be written as:
- Is there a course, X, that Turing teaches?
- The variable X stands for an object that the questioner does not know about yet.
- To answer the question, Prolog has to find out the value of X, if it exists.
- As long as we do not know the value of a variable it is said to be *unbound.*
- When a value is found, the variable is said to *bound* to that value.
- The name of a variable must begin with a capital letter or an underscore character, "_".

Variables 2

- To ask Prolog to find the course that Turing teaches, enter this: ?- *lectures(turing, Course).*

Course = 9020 ← output from Prolog

- To ask which course(s) Prof. Codd teaches, we may ask, ?- *lectures(codd , Course).*

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

If Prolog can tell that there are no more solutions, it just gives you the ?- prompt for a new query, as here. If Prolog can't tell, it will let you type ; again, and then if there is no further solution, report false.

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

Unification, Evaluation Order, and Backtracking

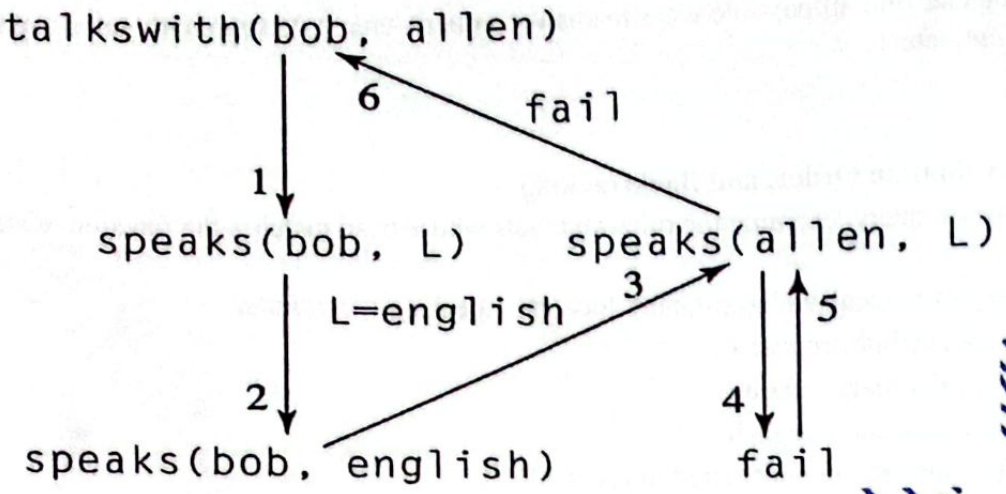
- To answer a query, examine the rules and facts whose head matches the function in the query.
- e.g.,: to solve speaks(Who,russian). look at – speaks(allen, russian).
  - speaks(bob, english).
  - speaks(mary, russian).
  - speaks(mary, english).
- Possible solutions are considered in order.
- Instantiating Who with allen generates the first match (since the second argument matches the query).

?-talkswith(bob, allen)

```
speaks(allen,russian).
speaks(bob, english).
speaks(mary, russian).
speaks(mary, english).
talkswith(P1,P2) :-
speaks(P1,L),
speaks(P2,L),
P1\=P2.
```

- *Instantiate and unify* variables in the rule with facts; *i.e.*, find values for P1, P2, and L that satisfy all 3 subgoals: P1 = bob, P2 = allen
- Can we find a value of L?
- Solve subgoals left to right in rule, work top to bottom in facts
- In case of failure, back-track to nearest subgoal

Attempting to Satisfy the Query talkswith (bob, allen)



Conjunctions of Goals in Queries

- How do we ask, "Does Turing teach Fred"?
- This means finding out if Turing lectures in a course that Fred studies.
  ?- *lectures(turing, Course), studies(fred, Course).*
- i.e. "Turing lectures in course, Course and Fred studies (the same) Course".
- The question consists of two *goals*.
- To answer this question, Prolog must find a single value for Course that satisfies both goals.
- Read the comma, ",", as **and**.
- However, note that Prolog will evaluate the two goals left-to-right. In pure logic, P1 ∧ P2 is the same as P2 ∧ P1. In Prolog, there is the practical consideration of which goal should be evaluated first – the code might be more efficient one way or the other. In particular, in "P1, P2", if P1 fails, then P2 does not need to be evaluated at all. This is sometimes referred to as a "conditional-and".

Disjunctions of Goals in Queries

- What about **or** (i.e. disjunction)? It turns out explicit **or**s aren't needed much in Prolog There is a way to write **or**: (";")
- The reason **or**s aren't needed much is that □ head :- body1.
  head :- body2. has the
  same effect as head :-
  body1 ; body2.
- Avoid using ; if you can, at least until you have learned how to manage without it. While some uses of ; are harmless, others can make your code hard to follow.
- To reinforce this message, you will be **penalised** if you use the **or** operator ; in the first

(Prolog) assignment in COMP9414. This prohibition means you can't use the ... -> ... ; ... construct either, in the first assignment. The ... -> ... ; ... construct is not taught in COMP9414, but in the past, some people have found out about it.

Backtracking in Prolog

- Who does Codd teach?
- ?- *lectures(codd, Course), studies(Student, Course).*

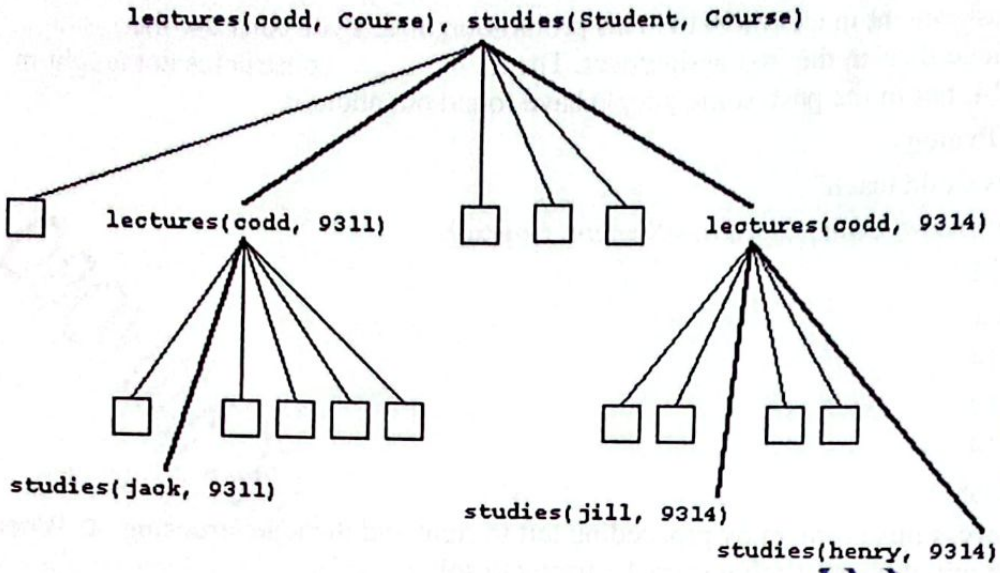
Course = 9311
Student = jack ;
Course = 9314
Student = jill ;
Course = 9314
Student = henry ;

- Prolog solves this problem by proceeding left to right and then *backtracking.* □ When given the initial query, Prolog starts by trying to solve

lectures(codd, Course)

- There are six lectures clauses, but only two have codd as their first argument.
- Prolog uses the first clause that refers to codd: lectures(codd, 9311).
- With Course = 9311, it tries to satisfy the next goal, studies(Student, 9311).
- It finds the fact studies(jack, 9311). and hence the first solution: (Course = 9311, Student = jack)
- After the first solution is found, Prolog retraces its steps up the tree and looks for alternative solutions.
- First it looks for other students studying 9311 (but finds none).
- Then it
  - backs up
  - rebinds Course to 9314,
  - goes down the lectures(codd, 9314) branch
  - tries studies(Student, 9314), finds the other two solutions: (Course = 9314, Student = jill) and (Course = 9314, Student = henry).

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



Clause Syntax

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

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

Tracing Execution

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

?- *trace*.

true.

[trace] ?- *more_advanced(henry, fred).*	
Call: more_advanced(henry, fred) ? *	bind S1 to henry, S2 to fred
Call: year(henry, _L205) ?	test 1st goal in body of rule
Exit: year(henry, 4) ?	succeeds, binds Year1 to 4
Call: year(fred, _L206) ?	test 2nd goal in body of rule
Exit: year(fred, 1) ?	succeeds, binds Year2 to 1
^ Call: 4>1 ?	test 3rd goal: Year1 > Year2
^ Exit: 4>1 ?	succeeds succeeds
Exit: more_advanced(henry, fred) ?	
true.	
[debug] ?- *notrace.*	

* The ? is a prompt. Press the return key at end of each line of tracing. Prolog will echo the <return> as creep, and then print the next line of tracing. The "creep"s have been removed in the table above, to reduce clutter.

true., false., or true

- Sometimes, Prolog says true instead of true. (i.e. no full-stop after true).
- Prolog does this when it believes it may be able to prove that the query is true in more than one way (and there are no variables in the query, that it can report bindings for).
- Example: suppose we have the following facts and rule: bad_dog(fido). bad_dog(Dog) :- bites(Dog, Person),
  is_person(Person),
  is_dog(Dog). bites(fido,
  postman).
  is_person(postman).
  is_dog(fido).

  There are two ways to prove bad_dog(fido): (a) it's there as a fact; and (b) it can be proven using the bad_dog rule:

  ?- *bad_dog(fido).*

  true ; true.

  The missing full-stop prompts us to type ; if we want to check for another proof. The true that follows means that a second proof *was* found. Alternatively, we can just press the "return" key if we are not interested in whether there is another proof.

Library Database Example

- A database of books in a library contains facts of the form : book(CatalogNo, Title, author(Family, Given)).
- libmember(MemberNo, name(Family, Given), Address).
- loan(CatalogNo, MemberNo, BorrowDate, DueDate).
- A member of the library may borrow a book.
- A "loan" records:

- the catalogue number of the book
- the number of the member
- the date on which the book was borrowed
- the due date

- Dates are stored as structures:
  date(Year, Month, Day)
- e.g. date(2022, 6, 16) represents 16 June 2022.
- which books has a member borrowed?

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

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

Comparing Two Terms

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

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

- This rule has three clauses: in any given case, only one clause is appropriate. They are tried in the given order.
- This is how disjunction (or) is often achieved in Prolog. In effect, we are saying that the first date is later than the second date if Day1 > Day2 and the Y and M are the same, **or** if the Y is the same and Month1 > Month2, **or** if Year1 > Year2.
- *Footnote*: if the year and month are the same, then the heads of all three rules match, and so, while the first rule is the appropriate one, all three will be tried in the course of backtracking. However, the condition "Month1 > Month2" in the second rule means that it will fail in this case, and similarly for the third rule.
- In the code for later, again we are using the comparison operator ">"


```
% Facts
book(101, 'Introduction to Prolog', author('Smith', 'John')).
book(102, 'Logic Programming Basics', author('Jones', 'Alice')).
book(103, 'Artificial Intelligence', author('Brown', 'David')).

libmember(001, name('Johnson', 'Mary'), '123 Main Street').
```

```
libmember(002, name('Wilson', 'Bob'), '456 Oak Avenue').
libmember(003, name('Taylor', 'Emily'), '789 Elm Lane').

loan(101, 001, date(2022, 6, 1), date(2022, 7, 1)).
loan(102, 002, date(2022, 5, 1), date(2022, 6, 1)).
loan(103, 003, date(2022, 7, 1), date(2022, 8, 1)).

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

% Comparing Two Dates
later(date(Y, Month1, Day1), date(Y, Month2, Day2)) :-
    Day1 > Day2,
    Month1 =:= Month2.
later(date(Year1, Month1, _), date(Year2, Month2, _)) :-
    Year1 > Year2.
later(date(Year, Month, _), date(Year, Month, Day1)) :-
    Day1 > 0.

% Finding overdue books
overdue_books(Title, CatalogNo, DueDate) :-
    loan(CatalogNo, _, _, DueDate),
    get_current_date(CurrentDate),
    later(CurrentDate, DueDate),
    book(CatalogNo, Title, _).

% Example usage
% To find books borrowed by a member:
% ?- borrowed('Johnson', Title, CatalogNo).
% This will find books borrowed by the member with the family name 'Johnson'.

% To find overdue books:
% ?- overdue_books(Title, CatalogNo, DueDate).
% This will find books that are overdue based on the current date.
```

- More complex arithmetic expressions can be arguments of comparison operators - e.g. X + Y >= Z * W * 2.
- The available *numeric* comparison operators are:

Operator	Meaning	Syntax
>	greater than	Expression1 > Expression2
<	less than	Expression1 < Expression2
>=	greater than or equal to	Expression1 >= Expression2
=<	less than or equal to	Expression1 =< Expression2
=:=	equal to	Expression1 =:= Expression2
=\=	not equal to	Expression1 =\= Expression2

- All these numerical comparison operators evaluate both their arguments. That is, they evaluate Expression1 and Expression2.

Overdue Books

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

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

Due Date

- Assume the loan period is one month:

```
due_date(date(Y, Month1, D),
date(Y, Month2, D)) :-
        Month1 < 12,
            Month2 is Month1 + 1.
due_date(date(Year1, 12, D),
date(Year2, 1, D)) :-
Year2 is Year1 + 1.
```

The is operator

- The right hand argument of is must be an arithmetic expression that can be evaluated right now (no unbound variables).
- This expression is evaluated and bound to the left hand argument.

- **"is"** is not a C-style assignment statement:
  - X is X + 1 won't work! o except via backtracking, variables can only be bound once, using is or any other way
- "=" does not cause evaluation of its arguments:

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

- Use **is** if and only if you need to evaluate something: X is 1 BAD! - nothing to evaluate
  X = 1 GOOD!
- To reinforce the point about the meaning of **is**, you will be penalised in the first Prolog assignment if you use it where it is not needed.

Order of goals with is

- Order of goals matters with **is.**
  Variables on the RHS of is *must* be instantiated at the time the is goal is tried by Prolog.
  This is why the following example fails:
  ?- X is Y + 1, Y = 3.
  ERROR: is/2: Arguments are not sufficiently instantiated
  vs
  ?- Y = 3, X is Y + 1.
  Y = 3,
  X = 4.

is, = and =:=

- You can see the differences between these three Prolog constructs from the following example Prolog queries:

?- X =:= 3+2. ERROR: =:=/2: Arguments are not sufficiently instantiated	X is not currently bound, so can't be evaluated.
?- X = 3+2. X = 3+2.	= doesn't evaluate, so X is bound to 3+2.
?- X is 3+2. X = 5.	is *does* evaluate its right-hand side.
?- 4+1 is 3+2. false.	3+2 is evaluated to 5. 4+1 is not evaluated. So 4+1 is different from 5.
?- 4+1=3+2. false.	Neither side is evaluated by =. The two expressions are different.

?- 4+1 =:= 3+2. true.	Both sides are evaluated by =:=

= is used for matching, so a more appropriate use would be:

?- likes(mary, X) = likes(Y, pizza).

X = pizza,

Y = mary.

Write a prolog program to sum two numbers: Option 1:

```
Clause: sum(X,Y,
SUM):- SUM is X+Y.
Goal: sum(3,5,S).
S=8.
```

Option 2:

```
sum(X,Y):- sum is
X+Y, write(sum).
Goal: sum(3,5,S).
S=8.
```

Option 3:

```
Sum:- readint(X),
readint(Y), sum is
X+Y, write (sum).
Goal: sum(3,5,S).
S=8.
```

Recursive Programs

- Recursion in any language is a function that call itself until a given goal has been succeeded
- In prolog, recursion appear when a predicate contain a goal that refers to itself.
- There are two types:
  Tail Recursion: in which the recursive call is always made Justin the last step before the procedure exits.
  Examples:
  Write a prolog program to find the factorial of 5. i.e. 5!

```
Domain : I=Integer Predicate:
fact(I,I,I).
Clauses:
fact(I,F,F):-!. fact(N,F,R):- F1 is F*N, N1 is N-1,
fact(N1,F1,R).
```

Goal: ? - fact(5,I,F).
Output : F=120.
Non-Trail Recursion : is recursion in which the recursion is not the last step in the procedure call.

```
fact(0,1). fact(1,1).
fact(N,F):- N1 is N-1, fact(N1,F1),  F is N*F1.
```

Goal : ?- fact(5,F).
Output : F=120.

Write a prolog program to print number from 1-20.

```
count(20):-!.
count(X):- write (X), X1 is X+1, count(X1).
Goal: count(1).
Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Write a program to find the power of any number using tail recursion.

```
power(_, 0, P,P):-!. power(X, Y, Z,P):- Z1 is Z*X, Y1 is Y-1, power(X, Y1, Z1,P),!.
Goal : power(5, 2, 1,P) Output: P=25.
```

Using non-trail recursion

```
power(_, 0, 1):-!. power(X, Y,P):- Y1 is Y-1, power(X, Y1,P1), P is P1*X.
Goal : power(5, 2, P) Output: P=25.
```

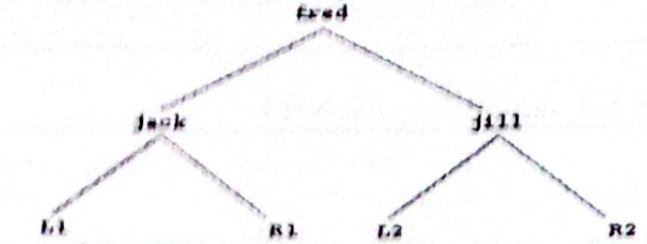
Write a program to read and write a number of characters until the input character is equal to '#'

```
Clause:
Repeat.
Repeat: repeat.
Typewriter :- repeat, readchar(C), write(C),nl, C is '#',!.
```


Binary Trees

- In the library database example, some complex terms contained other terms, for example, book contained name.
- The following term also contains another term, this time one similar to itself:
- tree(tree(L1, jack, R1), fred, tree(L2, jill, R2))

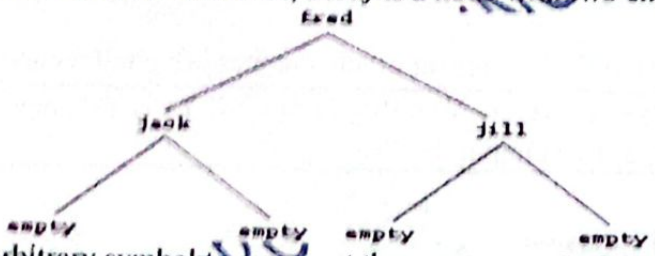
- The variables L1, L2, R1, and R2 should be bound to sub-trees (this will be clarified shortly).
- A structure like this could be used to represent a "binary tree" that looks like:



- Binary because each "node" has two branches (our backtrack tree before had many branches at some nodes)

Recursive Structures

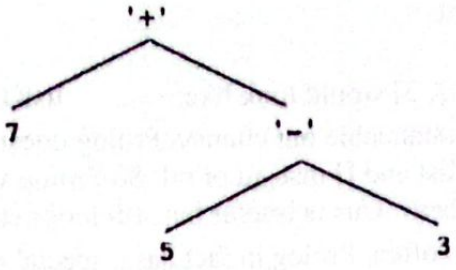
- A term that contains another term that has the same principal functor (in this case tree) is said to be recursive.
- Biological trees have leaves. For us, a *leaf* is a node with two empty branches:



- empty is an arbitrary symbol to represent the empty tree. In full, the tree above would be:
  tree(tree(empty, jack, empty), fred, tree(empty, jill, empty))
- Usually, we wouldn't bother to draw the empty nodes:

Another Tree Example

```
tree(tree(empty, 7, empty),
     2,
     tree(tree(empty, 5, empty),
          4,
          tree(empty, 3, empty)))
```



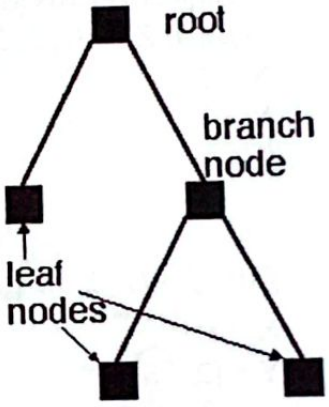
- A binary tree is either empty or contains some data and a left and right subtree that are also binary trees.
- In Prolog we express this as:

```
is_tree(empty).                    trivial branch
is_tree(tree(Left, Data, Right)) :-   recursive branch
is_tree(Left),      some_data(Data),
is_tree(Right).
```

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

The example at the heading "An Application of Lists", below, will show how the recursive branch and the trivial branch work together. However, you probably shouldn't try to look at it until we have studied lists.

- Let us define (or measure) the size of tree (i.e. number of nodes): tree_size(empty,0). tree_size(tree(L, _, R), Total_Size) :- tree_size(L, Left_Size), tree_size(R, Right_Size), Total_Size is Left_Size + Right_Size + 1.
- The size of an empty tree is zero.
- The size of a non-empty tree is the size of the left sub-tree plus the size of the right sub-tree plus one for the current tree node.
  - The data does not contribute to the total size of the tree.
- Recursive data structures need recursive programs. A recursive program is one that refers to itself, thus, tree_size contains goals that call for the tree_size of smaller trees **Lists**



Tree of size 5

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

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

- A list of numbers [1, 2, 3] would look like list(1, list(2, list(3, nil)))
- This notation is understandable but clumsy. Prolog doesn't actually recognise it, and in fact uses . instead of list and [] instead of nil. So Prolog would recognise .(1, .(2, .(3, []))) as a list of three numbers. This is briefer but still looks strange, and is hard to work with.
- Since lists are used so often, Prolog in fact has a special notation that encloses the list members in square brackets:

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

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

List Constructor |

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

?- *X = [1 | [2, 3]]*.
X = [1, 2, 3]
?- *Head = 1 , Tail = [2, 3], List = [Head | Tail]*.
List = [1, 2, 3].

Examples of Lists and Pattern Matching

?- *[X, Y, Z] = [1, 2, 3]*. Match the terms on either side of =
X = 1
Y = 2
Z = 3

[X | Y] = [1, 2, 3]. | separates head from tail of list. ?-
X = 1
Y = [2, 3]
So [First | Rest] is the usual way of writing (First, Rest) in Prolog

?- *[X] = [1]*.
X = 1
Y = []
The empty list is written as [] Lists "end" in an empty list!
Note that [1] is a list with one element

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

?- *[X, Y | Z] = [fred, jim, jill, mary]*.
X = fred Must be at least two elements Y = jim in the list
on the right.

Z = [jill, mary]

Complex List Matching

?- *[X | Y] = [[a, f(e)], [n, m, [2]]].*

X = [a, f(e)]
Y = [[n, m, [2]]]
Notice that Y is shown with an extra pair of brackets: Y is the tail of the entire list: [n, m, [2]] is the sole element of Y.

List Membership

- A term is a member of a list if ○ the term is the same as the head of the list, or ○ the term is a member of the tail of the list.
- In Prolog:

```
                                    trivial branch:
member(X, [X | _]).
                                                a rule with a head but no body
member(X, [_ | Y]) :-    recursive branch
    member(X, Y).
```

- The first rule has the same effect as: member(X, [Y|_]):- X = Y.
  The form member(X, [X|_]) is preferred, as it avoids the extra calculation.
- Member is actually predefined in Prolog. It is a built-in predicate. There are quite a few built-in predicates in Prolog.

```
% length(List, LengthOfList)
% binds LengthOfList to the number of elements in List.
length([OnlyMember], Length) :-
        Length = 1.
length([First | Rest], Length) :-
 length(Rest, LengthOfRest),  Length is
LengthOfRest + 1.
```

This works, but involves an unnecessary unification. It is better for the base case to be
length([OnlyMember], 1).
In effect, we take the original version of the base case, and replace Length, in the head of the rule, with the thing that Length is = to. Programmers who fail to do this are usually still thinking procedurally.

Programming Principles for Recursive Structures

- Only deal with one element at a time.

- Believe that the recursive program you are writing has already been written. In the definition of member, we are already assuming that we know how to find a member in the tail.
- Write definitions, not programs!
    - If you are used to writing programs for conventional languages, then you are used to giving instructions on how to perform certain operations.
    - In Prolog, you define relationships between objects and let the system do its best to construct objects that satisfy the given relationship.

Concatenating Two Lists

- Suppose we want to take two lists, like [1, 3] and [5, 2] and concatenate them to make [1, 3, 5, 2]
- The header comment is:

% concat(List1, List2, Concat_List1_List2)
% Concat_List1_List2 is the concatenation of List1 & List2 There are two rules:

- First, the trivial branch: concat([], List2, List2).
- Next, the recursive branch:
  concat([Item | Tail1], List2, [Item | Concat_Tail1_List2]) :- concat(Tail1, List2, Concat_Tail1_List2).
- For example, consider
  ?- concat([1], [2], [1, 2]).
  By the recursive branch:
- concat([1 | []], [2], [1 | [2]]) :-concat([], [2], [2]). and concat([], [2], [2]) holds because of the trivial branch.
- The entire program is:

```
% concat(List1, List2, Concat_List1_List2):
%  Concat_List1_List2 is the concatenation of List1 & List2
concat([], List2, List2). concat([Item | Tail1], List2, [Item |
Concat_Tail1_List2]) :-      concat(Tail1, List2,
Concat_Tail1_List2).
```

An Application of Lists

- Find the total cost of a list of items:

  Cost data:

  cost(cornflakes, 230).
  cost(cocacola, 210).
  cost(chocolate, 250).
  cost(crisps, 190).

- Rules:

```
total_cost([], 0).              % trivial branch
total_cost([Item|Rest], Cost) :-  % recursive branch
cost(Item, ItemCost),    total_cost(Rest,
CostOfRest),    Cost is ItemCost + CostOfRest.
```

Sample query:

?- *total_cost([cornflakes, crisps], X).*

X = 420

Tracing total_cost ?-

trace.

```
true.
[trace]  ?- total_cost([cornflakes, crisps], X).
   Call: (7) total_cost([cornflakes, crisps], _G290) ? creep
   Call: (8) cost(cornflakes, _L207) ? creep
   Exit: (8) cost(cornflakes, 230) ? creep
   Call: (8) total_cost([crisps], _L208) ? creep
   Call: (9) cost(crisps, _L228) ? creep
   Exit: (9) cost(crisps, 190) ? creep
   Call: (9) total_cost([], _L229) ? creep
   Exit: (9) total_cost([], 0) ? creep
^  Call: (9) _L208 is 190+0 ? creep
^  Exit: (9) 190 is 190+0 ? creep
   Exit: (8) total_cost([crisps], 190) ? creep
^  Call: (8) _G290 is 230+190 ? creep
^  Exit: (8) 420 is 230+190 ? creep
   Exit: (7) total_cost([cornflakes, crisps], 420) ? creep

X = 420

[debug]  ?- notrace.
```

Modifying total_cost

This is an *optional* homework exercise.

What happens if we change the recursive branch rule for total_cost as shown below?

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

The second and third lines have been swapped around.
You'll find that the rule still works. Try tracing the new version of this rule, work out what happens differently.
Which version do you find easier to understand? Why do think this is the case?

Another list-processing procedure

- The next procedure removes duplicates from a list.
- It has *three rules*. This is an example of a common list-processing *template*.
- Algorithm: ○ If the list is empty, there's nothing to do.
  ○ If the first item of the list is a member of the rest of the list, then discard it, and remove duplicates from the rest of the list. ○ Otherwise, keep the first item, and again, remove any duplicates from the rest of the list.

```
% remove_dups(+List, -NewList).
% New List isbound to List, but with duplicate items removed. remove_dups([],
[]).
remove_dups([First | Rest], NewRest) :-
member(First, Rest),    remove_dups(Rest,
NewRest). remove_dups([First | Rest], [First |
NewRest]) :-    not(member(First, Rest)),
remove_dups(Rest, NewRest).
```

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

X = [2, 1, 3, 4] ;

false.
```

- Note the use of not to negate a condition. An alternative to not is \+.

Singleton Varinbles

- If Prolog finds a variable name that you only use once in a rule, it assumes that it may be a spelling mistake, and issues a **Warning** about a "singleton variable" when you load the code:

```
%prolog -q -s mycode.pl
Warning: .../mycode.pl:4:
                Singleton variables: [Item]
```

The -q means "quiet" - i.e. don't print the SWI Prolog welcome message. This way, any warnings are easier to notice.

- Here is the code that produced this (with line numbers added): 1 % count(Item, List, Count) counts the number of times the 2 % Item occurs in the List, and binds Count to that number.

```
3
4 count(Item, [], 0).
5 count(Item, [Item | Rest], Count) :- 6
  count(Item, Rest, RestCount), 7    Count is
  RestCount + 1.
    8       count(Item, [Other | Rest],
Count) :-
    9       not(Item = Other),    10
count(Item, Rest, Count).
```

To suppress the warning, put an _ in front of the word Item on line 4 (only). This makes it "don't care" variable. Check for the possible spelling error, first!

```
4 count(_Item, [], 0).
```

Controlling Execution

The Cut Operator/function (!)

- Sometimes we need a way to prevent Prolog finding all solutions, i.e. a way to stop backtracking.
- The cut operator, written !, is a built-in goal that prevents backtracking.

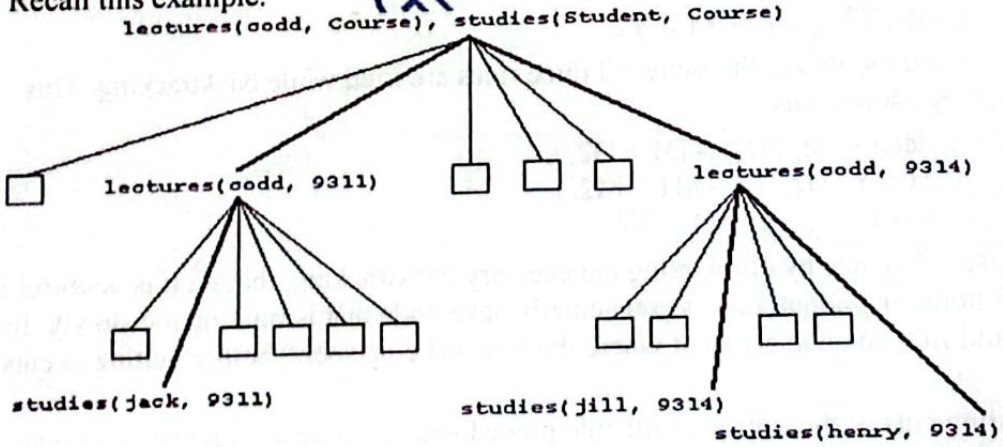
- It turns Prolog from a nice declarative language into a hybrid monster.
- Use cuts sparingly and with a sense of having sinned.

```
% Using cut
no(5):-!.
no(1).
no(10).

| ?- no(X).

X = 5

yes
```

Using cut in the end of the rule.

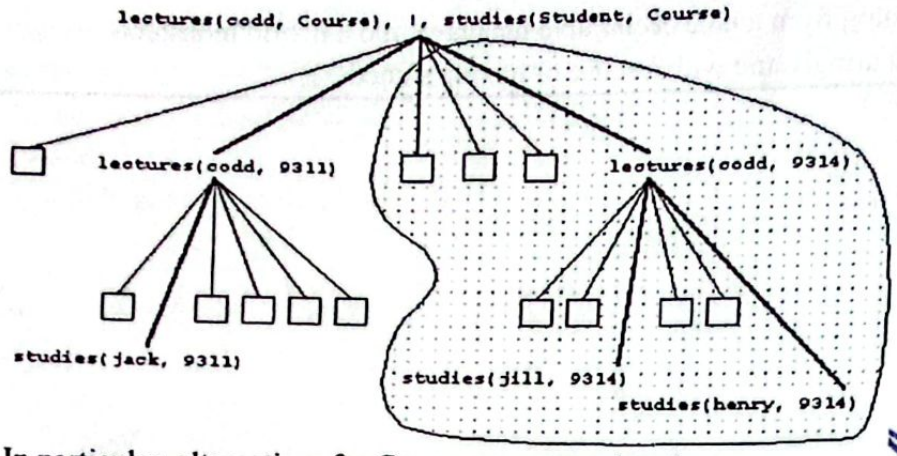
```
a(10).
a(20).
b(a).
b(c).
c(X,Y):- a(X), b(Y), !.

| ?- c(X,Y).
X = 10
Y = a

yes
```

Recall this example:



Cut Prunes the Search Tree

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



- In particular, alternatives for Course are not explored.

Cut Prunes the Search Tree 2

- Another example: using the facts03 database, try ?- *lectures(codd, X).* X = 9311 ; X = 9314.

?- *lectures(codd, X), !.*

X = 9311.

- The cut in the second version of the query prevents Prolog from backtracking to find the second solution.

Using cuts in later to improve efficiency

Recall the code for later:

```
later(date(Y, M, D1), date(Y, M, D2)) :- D1 > D2.
later(date(Y, M1, _), date(Y, M2, _)) :- M1 > M2.
later(date(Y1, _, _), date(Y2, _, _)) :- Y1 > Y2.
```

We note that if year and month are the same, all three rules are tried while backtracking. This could be prevented by adding cuts:

```
later(date(Y, M, D1), date(Y, M, D2)) :- D1 > D2, !.
later(date(Y, M1, _), date(Y, M2, _)) :- M1 > M2, !.
later(date(Y1, _, _), date(Y2, _, _)) :- Y1 > Y2.
```

This would increase efficiency by eliminating unnecessary backtracking, though it is doubtful if it would be worth bothering about, unless you actually have code that is running too slowly. In that case you should first do an analysis of where the time is being spent, before putting in cuts everywhere!

In other cases, adding cuts of this sort to multi-rule procedures might be a useful (if lazy) way of ensuring that only one rule is used in a particular case. Unless it makes the code very clumsy, it is better to use and rely on "condition" goals in each rule (like M1 > M2 in the second rule for later) to specify the case in which it is appropriate. More examples of this are below.

Another cut example

- max, without cut:

```
% max(A, B, C) binds C to the larger of A and B.
max(A, B, A) :-   A > B.
max(A, B, B) :- A =< B.
```

- max, with cut:

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

- The first version has a negated test in the second rule (=< vs >). The second version substitutes a cut in the first rule for the negated test.
- Remember, no cuts in the first assignment unless they are essential! Hint: the first assignment can be done without cuts.

PROGRAMS