Programming Paradigms

Lecture 11. Lists and arithmetic in Prolog

Outline

- Clarification on terms
- Syntax for lists
- Implementing recursive predicates on lists
- Arithmetic in Prolog

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To,              ) :- connected(From, Next).
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To,                    )
  :-
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To,                          )
  :- connected(From, Next),
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To,                          )
  :- connected(From, Next), path(Next, To, Path).
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To, go(From, Next, Path))
  :- connected(From, Next), path(Next, To, Path).
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To, go(From, Next, Path))
  :- connected(From, Next), path(Next, To, Path).

?- path(1, 1, Path)
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To, go(From, Next, Path))
  :- connected(From, Next), path(Next, To, Path).

?- path(1, 1, Path)
Path = stop
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To, go(From, Next, Path))
  :- connected(From, Next), path(Next, To, Path).

?- path(1, 2, Path)
Path = go(1, 2)
```

Clarification: complex terms vs predicate calls

```
connected(1, 2).
connected(2, 3).

path(From, From, stop).
path(From, To, go(From, To)) :- connected(From, Next).
path(From, To, go(From, Next, Path))
  :- connected(From, Next), path(Next, To, Path).

?- path(1, 3, Path)
Path = go(1, 2, go(2, 3))
```

List syntax

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

List syntax

```
[1, 2, 3]

[hello, X, flight(boston, To), 10]
```

List syntax

```
[1, 2, 3]

[hello, X, flight(boston, To), 10]

[[], hello, X, [1, 2]]
```

List syntax: the | operator

```
?- [Head|Tail] = [1, 2, 3]
```

List syntax: the | operator

```
?- [Head|Tail] = [1, 2, 3]
Head = 1,
Tail = [2, 3]
```

List syntax: the | operator

```
?- [Head|Tail] = [1, 2, 3]
Head = 1,
Tail = [2, 3]

?- [First,Second|Tail] = [1, 2, 3, 4, 5]
```

List syntax: the | operator

```
?- [Head|Tail] = [1, 2, 3]
Head = 1,
Tail = [2, 3]

?- [First,Second|Tail] = [1, 2, 3, 4, 5]
First = 1,
Second = 2,
Tail = [3, 4, 5]
```

Anonymous variable

```
?- [X1,X2,X3,X4|Tail] = [1, 2, 3, 4, 5]
X1 = 1,
X2 = 2,
X3 = 3,
X4 = 4,
Tail = [5]
```

Anonymous variable

```
?- [_,X2,_,X4|_] = [1, 2, 3, 4, 5]
X2 = 2,
X4 = 4
```

`member` predicate

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

`member` predicate

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

member predicate

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

member predicate

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

?- member(c, [a, b, c, d]).
true
```

member predicate

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

?- member(c, [a, b, c, d]).
true

?- member(X, [a, b, c, d]).
```

member predicate

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

?- member(c, [a, b, c, d]).
true

?- member(X, [a, b, c, d]).
X = a
```

member predicate

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

?- member(c, [a, b, c, d]).
true

?- member(X, [a, b, c, d]).
X = a
X = b
```

member predicate

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

?- member(c, [a, b, c, d]).
true

?- member(X, [a, b, c, d]).
X = a
X = b
X = c
```

member predicate

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

?- member(c, [a, b, c, d]).
true

?- member(X, [a, b, c, d]).
X = a
X = b
X = c
X = d
```

Arithmetic in Prolog

```
?-  8  is  6+2.
yes
```

Arithmetic in Prolog

```
?-  8  is  6+2.
yes

?-  12  is  6*2.
yes

?-  -2  is  6-8.
yes

?-  3  is  6/2.
yes
```

Arithmetic in Prolog

```
?-  8  is  6+2.
yes

?-  12  is  6*2.
yes

?-  -2  is  6-8.
yes

?-  3  is  6/2.
yes

?-  1  is  mod(7,2).
yes
```

Arithmetic in Prolog

```
?-  X  is  6+2.
X = 8
```

Arithmetic in Prolog

```
?-  X  is  6+2.
X = 8

?-  8  is  6+Y.
```

Arithmetic in Prolog

```
?-  X  is  6+2.
X = 8

?-  8  is  6+Y.

Arguments are not sufficiently instantiated
In:
   [1] 8 is 6+_1634
```

Arithmetic in Prolog

```
?-  X  is  6+2.
X = 8

six_plus(X, Y) :- X is 6+Y.
```

Arithmetic in Prolog

```
?-  X  is  6+2.
X = 8

six_plus(X, Y) :- X is 6+Y.

?-  six_plus(X, 2).
X = 8
```

Length of a list

```
length([], 0).
```

Length of a list

```
length([], 0).
length([_|T], N) :-
```

Length of a list

```
length([], 0).
length([_|T], N) :- length(T, M),
```

Length of a list

```
length([], 0).
length([_|T], N) :- length(T, M), N is M+1.
```

Length of a list

```
length([], 0).
length([_|T], N) :- length(T, M), N is M+1.

?- length([a, b, c], X)
X = 3
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- binary(X), length(X, 3).
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- binary(X), length(X, 3).
X = [0, 0, 0]
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- binary(X), length(X, 3).
X = [0, 0, 0]
X = [0, 0, 1]
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- binary(X), length(X, 3).
X = [0, 0, 0]
X = [0, 0, 1]
(loops indefinitely)
```

Debugging with **`trace`**

```
?- trace, binary(X), length(X, 3).
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
 Exit:length([0, 0, 0], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
 Exit:length([0, 0, 0], 3)
X = [0, 0, 0]
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
 Exit:length([0, 0, 0], 3)
X = [0, 0, 0]
```

```
Exit:binary([0, 0, 1])
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
 Exit:length([0, 0, 0], 3)
X = [0, 0, 0]
```

```
 Exit:binary([0, 0, 1])
 Exit:length([0, 0, 1], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
 Exit:length([0, 0, 0], 3)
X = [0, 0, 0]
```

```
 Exit:binary([0, 0, 1])
 Exit:length([0, 0, 1], 3)
X = [0, 0, 1]
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
 Exit:length([0, 0, 0], 3)
X = [0, 0, 0]
```

```
 Exit:binary([0, 0, 1])
 Exit:length([0, 0, 1], 3)
X = [0, 0, 1]
 Exit:binary([0, 0, 0, 0])
 Fail:length([0, 0, 0, 0], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
 Exit:binary([0])
 Fail:length([0], 3)
 Exit:binary([1])
 Fail:length([1], 3)
 Exit:binary([0, 0])
 Fail:length([0, 0], 3)
 Exit:binary([0, 1])
 Fail:length([0, 1], 3)
 Exit:binary([0, 0, 0])
 Exit:length([0, 0, 0], 3)
X = [0, 0, 0]
```

```
 Exit:binary([0, 0, 1])
 Exit:length([0, 0, 1], 3)
X = [0, 0, 1]
 Exit:binary([0, 0, 0, 0])
 Fail:length([0, 0, 0, 0], 3)
 Exit:binary([0, 0, 0, 1])
 Fail:length([0, 0, 0, 1], 3)
```

Debugging with **trace**

```
?- trace, binary(X), length(X, 3).
   Exit:binary([0])
   Fail:length([0], 3)
   Exit:binary([1])
   Fail:length([1], 3)
   Exit:binary([0, 0])
   Fail:length([0, 0], 3)
   Exit:binary([0, 1])
   Fail:length([0, 1], 3)
   Exit:binary([0, 0, 0])
   Exit:length([0, 0, 0], 3)
X = [0, 0, 0]
```

```
   Exit:binary([0, 0, 1])
   Exit:length([0, 0, 1], 3)
X = [0, 0, 1]
   Exit:binary([0, 0, 0, 0])
   Fail:length([0, 0, 0, 0], 3)
   Exit:binary([0, 0, 0, 1])
   Fail:length([0, 0, 0, 1], 3)
   Exit:binary([0, 0, 0, 0, 0])
   Fail:length([0, 0, 0, 0, 0], 3)
   ...
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- length(X, 3), binary(X).
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- length(X, 3), binary(X).
X = [0, 0, 0]
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- length(X, 3), binary(X).
X = [0, 0, 0]
X = [0, 0, 1]
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- length(X, 3), binary(X).
X = [0, 0, 0]
X = [0, 0, 1]
...
X = [1, 1, 1]
```

Standard `length`

```
binary([0]).
binary([1]).
binary([0|T]) :- binary(T).
binary([1|T]) :- binary(T).

?- length(X, 3), binary(X).
X = [0, 0, 0]
X = [0, 0, 1]
...
X = [1, 1, 1]
false
```

Appending lists

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

Appending lists

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

Appending lists

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

Appending lists

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

?- append([1,2], [3,4], R)
```

Appending lists

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

?- append([1,2], [3,4], R)
R = [1,2,3,4]
```

Appending lists

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

?- append([1,2], [3,4], R)
R = [1,2,3,4]

?- append(X, Y, [1, 2, 3])
```

Appending lists

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

?- append([1,2], [3,4], R)
R = [1,2,3,4]

?- append(X, Y, [1, 2, 3])
X = [],          Y = [1, 2, 3]
```

Appending lists

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

?- append([1,2], [3,4], R)
R = [1,2,3,4]

?- append(X, Y, [1, 2, 3])
X = [],          Y = [1, 2, 3]
X = [1],         Y = [1, 2]
```

Appending lists

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

?- append([1,2], [3,4], R)
R = [1,2,3,4]

?- append(X, Y, [1, 2, 3])
X = [],          Y = [1, 2, 3]
X = [1],         Y = [1, 2]
X = [1, 2],      Y = [1]
```

Appending lists

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

?- append([1,2], [3,4], R)
R = [1,2,3,4]

?- append(X, Y, [1, 2, 3])
X = [],          Y = [1, 2, 3]
X = [1],         Y = [1, 2]
X = [1, 2],      Y = [1]
X = [1, 2, 3], Y = []
```

Reverse: naive implementation

```
reverse([], []).
```

Reverse: naive implementation

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

Reverse: naive implementation

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

Reverse: naive implementation

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

Reverse: naive implementation

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

1. `reverse([a, b, c], Z)`

Reverse: naive implementation

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

1. `reverse([a, b, c], Z)`
2. `reverse([b, c], X1), append(X1, [a], Z)`

Reverse: naive implementation

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

1. `reverse([a, b, c], Z)`
2. `reverse([b, c], X1), append(X1, [a], Z)`
3. `reverse([c], X2), append(X2, [b], X1), append(X1, [a], Z)`

Reverse: naive implementation

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

```
1.  reverse([a, b, c], Z)
2.  reverse([b, c], X1), append(X1, [a], Z)
3.  reverse([c], X2), append(X2, [b], X1), append(X1, [a], Z)
4.  reverse([], X3), append(X3, [c], X2), append(X2, [b], X1), append(X1, [a], Z
```

Reverse: naive implementation

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

```
1.  reverse([a, b, c], Z)
2.  reverse([b, c], X1), append(X1, [a], Z)
3.  reverse([c], X2), append(X2, [b], X1), append(X1, [a], Z)
4.  reverse([], X3), append(X3, [c], X2), append(X2, [b], X1), append(X1, [a], Z
5.  X3 = []
    append([], [c], X2), append(X2, [b], X1), append(X1, [a], Z)
```

Reverse: naive implementation

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

```
1.  reverse([a, b, c], Z)
2.  reverse([b, c], X1), append(X1, [a], Z)
3.  reverse([c], X2), append(X2, [b], X1), append(X1, [a], Z)
4.  reverse([], X3), append(X3, [c], X2), append(X2, [b], X1), append(X1, [a], Z
5.  X3 = []
    append([], [c], X2), append(X2, [b], X1), append(X1, [a], Z)
6.  X2 = [c]
    append([c], [b], X1), append(X1, [a], Z)
```

Reverse: naive implementation

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

```
1.  reverse([a, b, c], Z)
2.  reverse([b, c], X1), append(X1, [a], Z)
3.  reverse([c], X2), append(X2, [b], X1), append(X1, [a], Z)
4.  reverse([], X3), append(X3, [c], X2), append(X2, [b], X1), append(X1, [a], Z
5.  X3 = []
    append([], [c], X2), append(X2, [b], X1), append(X1, [a], Z)
6.  X2 = [c]
    append([c], [b], X1), append(X1, [a], Z)
7.  X1 = [c, b]
    append([c, b], [a], Z)
```

Reverse: naive implementation

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

```
1.  reverse([a, b, c], Z)
2.  reverse([b, c], X1), append(X1, [a], Z)
3.  reverse([c], X2), append(X2, [b], X1), append(X1, [a], Z)
4.  reverse([], X3), append(X3, [c], X2), append(X2, [b], X1), append(X1, [a], Z
5.  X3 = []
    append([], [c], X2), append(X2, [b], X1), append(X1, [a], Z)
6.  X2 = [c]
    append([c], [b], X1), append(X1, [a], Z)
7.  X1 = [c, b]
    append([c, b], [a], Z)
8.  Z = [c, b, a]
```

Reverse with accumulator parameter

```
reverse([], Acc, Acc).
reverse([Head|Tail], Acc, Result)
    :- reverse(Tail, [Head|Acc], Result).
```

Reverse with accumulator parameter

```
reverse([], Acc, Acc).
reverse([Head|Tail], Acc, Result)
    :- reverse(Tail, [Head|Acc], Result).
```

1. `reverse([a, b, c], [], Z)`

Reverse with accumulator parameter

```
reverse([], Acc, Acc).
reverse([Head|Tail], Acc, Result)
    :- reverse(Tail, [Head|Acc], Result).
```

1. `reverse([a, b, c], [], Z)`
2. `reverse([b, c], [a], Z)`

Reverse with accumulator parameter

```
reverse([], Acc, Acc).
reverse([Head|Tail], Acc, Result)
    :- reverse(Tail, [Head|Acc], Result).
```

1. `reverse([a, b, c], [], Z)`
2. `reverse([b, c], [a], Z)`
3. `reverse([c], [b, a], Z)`

Reverse with accumulator parameter

```
reverse([], Acc, Acc).
reverse([Head|Tail], Acc, Result)
    :- reverse(Tail, [Head|Acc], Result).
```

1. `reverse([a, b, c], [], Z)`
2. `reverse([b, c], [a], Z)`
3. `reverse([c], [b, a], Z)`
4. `reverse([], [c, b, a], Z)`

Reverse with accumulator parameter

```
reverse([], Acc, Acc).
reverse([Head|Tail], Acc, Result)
    :- reverse(Tail, [Head|Acc], Result).
```

1. `reverse([a, b, c], [], Z)`
2. `reverse([b, c], [a], Z)`
3. `reverse([c], [b, a], Z)`
4. `reverse([], [c, b, a], Z)`
5. `Z = [c, b, a]`

Homework (self-study)

1. Read about **Declarative Debugging of Prolog Programs**
   https://www.metalevel.at/prolog/debugging
2. Read **Chapters 6 and 10** of Learn Prolog Now!
   http://www.let.rug.nl/bos/lpn/lpnpage.php?pagetype=html&pageid=lpn-htmlch1
3. Work through the **exercises** from both chapters, using SWISH
   https://swish.swi-prolog.org

What was the most unclear part of the lecture for you?

See Moodle

References

1. Learn Prolog Now!