CS2510
MODERN PROGRAMMING LANGUAGES

Logic Programming 4

Prof. Peter Edwards
p.edwards@abdn.ac.uk

Prolog Programming with Lists

- Because of their recursive definition, lists are naturally manipulated with recursive programs.

- Let's revisit the earlier `loop` program:
  - Enable it to build a list of values typed in at keyboard:

```
loop(L):-          % a loop to create a list L
    L = [X|Xs],    % List L has form [X|Xs]
    read(X),       % X is read from the standard Input (keyboard)
    X \= end,      % if input different from "end" then
    loop(Xs).      % build the tail recursively
loop([]).          % if input is "end", then List is empty.
```

An argument is required to return the value of the list built.

```
?- loop(MyList).
|: a.
|: b.
|: end.
MyList = [a,b]
```

SWI Prolog

UNIVERSITY OF ABERDEEN

Programming with Lists

- Its execution:

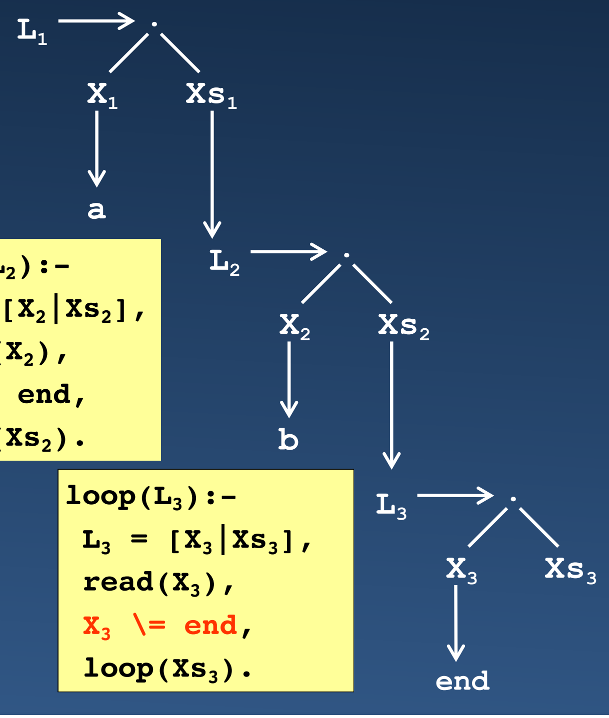
```
loop(L):-
  L = [X|Xs],
  read(X),
  X \= end,
  loop(Xs).
loop([]).
```

```
?- loop(MyList).
|: a.
|: b.
|: end.
```

```
loop(L1):-
 L1 = [X1|Xs1],
 read(X1),
 X1 \= end,
 loop(Xs1).
```

```
loop(L2):-
 L2 = [X2|Xs2],
 read(X2),
 X2 \= end,
 loop(Xs2).
```

```
loop(L3):-
 L3 = [X3|Xs3],
 read(X3),
 X3 \= end,
 loop(Xs3).
```



Programming with Lists

- Its execution:

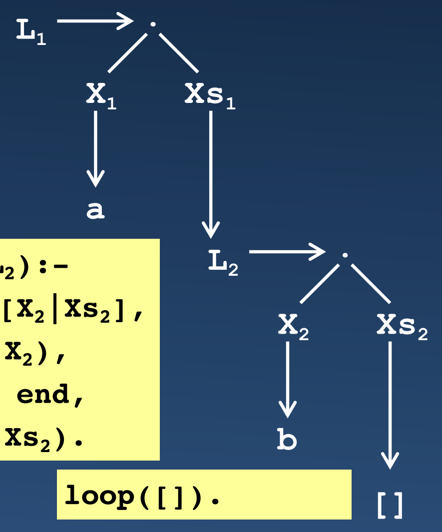
```
loop(L):-
  L = [X|Xs],
  read(X),
  X \= end,
  loop(Xs).
loop([]).
```

```
?- loop(MyList).
|: a.
|: b.
|: end.
MyList = [a,b]
```

```
loop(L1):-
 L1 = [X1|Xs1],
 read(X1),
 X1 \= end,
 loop(Xs1).
```

```
loop(L2):-
 L2 = [X2|Xs2],
 read(X2),
 X2 \= end,
 loop(Xs2).
```

```
loop([]).
```



Programming with Lists: `member()`

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

- To check if an element appears in a list:
  - Check if element is the head;
  - Otherwise, check if element appears in tail.
- This formulation is guaranteed to stop because lists are finite and the process breaks list apart.

- In *Prolog* (first formulation):

```
member(X,List):-      % X is a member of List
  List = [Y|Ys],      % if List is of form [Y|Ys]
  X = Y.              % and X is equal to (unifies with) Y
member(X,List):-      % otherwise (if not as above)
  List = [Y|Ys],      % break the list apart
  member(X,Ys).       % and check if X appears in Ys
```

Programming with Lists: `member()`

- Second formulation:

```
member(X,[Y|Ys]):-      % X is a member of [Y|Ys]
  X = Y.                % if X is equal to Y
member(X,[Y|Ys]):-      % otherwise (if not as above)
  member(X,Ys).         % check if X appears in Ys
```

- Third (final) formulation:

```
member(X,[X|_]).        % X is member if it is head of list
member(X,[_|Ys]):-      % otherwise (if not as above)
  member(X,Ys).         % check if X appears in Ys
```

Programming with Lists: `member()`

- Let's run the previous program:

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

```
?- member(3,[1,2,3]).
true
?-
```

The query `member(3,[1,2,3])` does not match the 1st clause!!

Prolog matches `member(3,[1,2,3])` with head of 2nd clause:

$$\texttt{member}(X_1,[\_|Ys_1]) \texttt{:-} \texttt{member}(X_1,Ys_1)$$

We thus get the values for the variables: $\{X_1/3, Ys_1/[2,3]\}$

Prolog now tries to prove $\texttt{member}(X_1,Ys_1)$ that is

`member(3,[2,3])`

The goal `member(3,[2,3])` does not match the 1st clause!!

Prolog matches `member(3,[2,3])` with head of 2nd clause:

$$\texttt{member}(X_2,[\_|Ys_2]) \texttt{:-} \texttt{member}(X_2,Ys_2)$$

We thus get the values for the variables: $\{X_2/3, Ys_2/[3]\}$

Prolog now tries to prove $\texttt{member}(X_2,Ys_2)$ that is

`member(3,[3])`

The query `member(3,[3])` matches the 1st clause!!

Programming with Lists: `member()`

- `member` code can be run in two ways:
  - To check if an element occurs in a list;
  - To obtain, the elements of a list, one at a time
- It depends on the query – the code is exactly the same!

- To obtain the elements of a list, we query:

  `member(Elem,[1,2,3])`
  - We then get in **`Elem`**, the different elements:

```
?- member(Elem,[1,2,3]).
Elem = 1 ? ;
Elem = 2 ? ;
Elem = 3 ? ;
false
?-
```

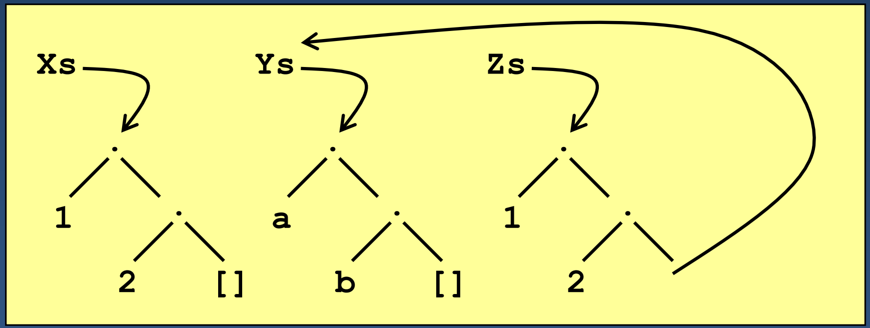
SWI Prolog

Can you trace this query?

Programming with Lists: `append()`

```
?- append([1,2], [a, b], Res).
Res = [1,2,a,b] ?
true
?-
```

- To concatenate two lists `Xs` and `Ys`:
  - Create a 3rd list `Zs` by copying all elements of `Xs`;
  - When we get to the end of `Xs`, put `Ys` as the tail of `Zs`.
- Diagrammatically:



UNIVERSITY OF ABERDEEN

Programming with Lists: `append()`

- We need a predicate with 3 arguments:
  - 1st argument is the list `Xs`
  - 2nd argument is the list `Ys`
  - 3rd argument is the concatenation **`Xs.Ys`** (in this order)
- The first list `Xs` will control the loop
- As the first list `Xs` is traversed, its elements are copied onto the 3rd argument (the concatenation).
- When we get an empty list, we stop and make `Ys` the value of the 3rd argument

Programming with Lists: `append()`

- First formulation:

```
append([],Ys,Zs):-            % if Xs is empty,
  Zs = Ys.                    %   then Zs is Ys
append([X|Xs],Ys,[Z|Zs]):-   % otherwise
  X = Z,                      %   copy X onto 3rd argument
  append(Xs,Ys,Zs).           %   carry on recursively
```

- A more compact format:

```
append([],Ys,Ys).             % if Xs empty, then Zs is Ys
append([X|Xs],Ys,[X|Zs]):-   % else copy X onto 3rd arg.
    append(Xs,Ys,Zs).         %   carry on recursively
```

SWI Prolog

Programming with Lists: `append()`

- Execution:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-
    append(Xs,Ys,Zs).
```

```
?- append([1,2],[a,b],Res).
Res = [1,2,a,b] ?
true
?-
```

aliasing: $\texttt{Res/[X}_1\texttt{|Zs}_1\texttt{]}$

$\texttt{append([X}_1\texttt{|Xs}_1\texttt{],Ys}_1\texttt{,[X}_1\texttt{|Zs}_1\texttt{]):-}$
$\quad\texttt{append(Xs}_1\texttt{,Ys}_1\texttt{,Zs}_1\texttt{).}$
$\texttt{\{X}_1\texttt{/1,Xs}_1\texttt{/[2],Ys}_1\texttt{/[a,b]\}}$

aliasing: $\texttt{Zs}_1\texttt{/[X}_2\texttt{|Zs}_2\texttt{]}$

$\texttt{append([X}_2\texttt{|Xs}_2\texttt{],Ys}_2\texttt{,[X}_2\texttt{|Zs}_2\texttt{]):-}$
$\quad\texttt{append(Xs}_2\texttt{,Ys}_2\texttt{,Zs}_2\texttt{).}$
$\texttt{\{X}_2\texttt{/2,Xs}_2\texttt{/[],Ys}_2\texttt{/[a,b]\}}$

aliasing: $\texttt{Zs}_2\texttt{/Ys}_3$

$\texttt{append([],Ys}_3\texttt{,Ys}_3\texttt{).}$
$\texttt{\{Ys}_3\texttt{/[a,b]\}}$

UNIVERSITY OF ABERDEEN

Programming with Lists: `append()`

- Execution:

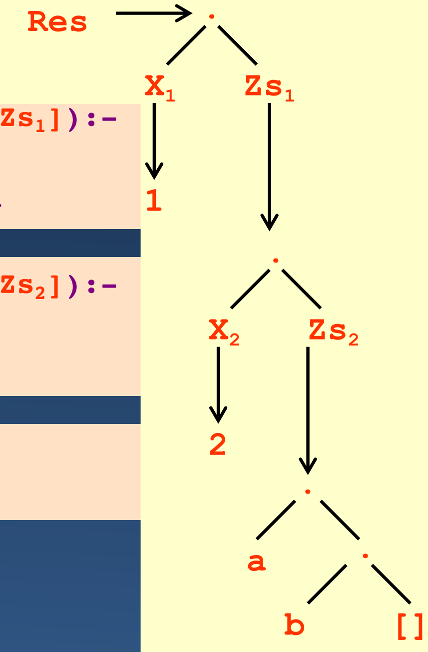
```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-
    append(Xs,Ys,Zs).
```

```
?- append([1,2],[a,b],Res).
Res = [1,2,a,b] ?
true
?-
```

$\mathtt{append([X_1|Xs_1],Ys_1,[X_1|Zs_1])}$:-
    $\mathtt{append(Xs_1,Ys_1,Zs_1).}$
$\mathtt{\{X_1/1,Xs_1/[2],Ys_1/[a,b]\}}$

$\mathtt{append([X_2|Xs_2],Ys_2,[X_2|Zs_2])}$:-
    $\mathtt{append(Xs_2,Ys_2,Zs_2).}$
$\mathtt{\{X_2/2,Xs_2/[\,],Ys_2/[a,b]\}}$

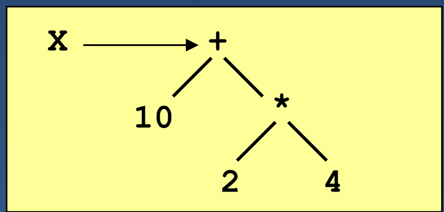
$\mathtt{append([\,],Ys_3,Ys_3).}$
$\mathtt{\{Ys_3/[a,b]\}}$



Arithmetic

- Arithmetic expressions are also trees:

```
?- X = 10 + (2 * 4).
X = 10 + (2 * 4) ?
true
? -
```



- *Prolog* does not treat arithmetic expressions differently, unless asked to do so.
- To evaluate arithmetic expressions, use the built-in “`is`” – as follows:

```
?- X is 10 + (2 * 4).
X = 18 ?
true
?-
```

Arithmetic

- The syntax of the built-in "**is**" is:

  `Variable is ArithmeticExpression`

- Examples:

```
Result is (10 * 7)
NewRes is OldRes + 1
```

- Variables that appear in the expression
  - Must be instantiated when expression is evaluated;
  - Otherwise execution error!

- Can delay the evaluation until all variables have value:

```
?- Exp = A + B, A = 3, B = 4, Res is Exp.
A = 3,
B = 4,
Exp = 3 + 4
Res = 7
```

SWI Prolog

UNIVERSITY OF ABERDEEN

Arithmetic: `length()`

```
?- length([a,b,c,d],Length).
Length = 4 ?
?- length([1,[2,3],4],L).
L = 3 ?
?-
```

- Find the size (number of elements) of a list.
- Simple recursive formulation:
  - The number of elements of the empty list is zero;
  - The number of elements of a list `[X|Xs]` is one plus the number of elements of `Xs`.

- Predicate **`length`** of arity 2:
  - 1st argument is the list;
  - 2nd argument is the size of the list.

Arithmetic: `length()`

- First formulation:

```
length(L,S):-                 % S is the length of list L
  L = [],                     % if L is an empty list
  S = 0.                      % its length S is zero
length(L,S):-                 % otherwise
  L = [X|Xs],                 % if L is a list [X|Xs]
  S is 1 + length(Xs,S).      % its length is 1 plus length of tail
```

CARE!
Prolog calls are **true** or **false**.
This expression does not make sense!

UNIVERSITY OF ABERDEEN

Arithmetic: `length()`

- Second formulation:

```
length(L,S):-           % S is the length of list L
  L = [],               %   if L is an empty list
  S = 0.                %   its length S is zero
length(L,S):-           % otherwise
  L = [X|Xs],           % if L is a list [X|Xs]
  length(Xs,1 + SXs).   % get the length SXs of tail Xs
```

CARE! This is just building a structure, not computing an expression! What about **S** ?

- Third formulation:

```
length(L,S):-       % S is the length of list L
  L = [],           %   if L is an empty list
  S = 0.            %   its length S is zero
length(L,S):-       % otherwise
  L = [X|Xs],       % if L is a list [X|Xs]
  S is 1 + SXs,     % length S is 1 plus length of tail
  length(Xs,SXs).   % get the length SXs of tail Xs
```

CARE! When this expression is evaluated, the value of **SXs** won't yet exist!

Arithmetic: `length()`

- Fourth formulation:

```
length(L,S):-       % S is the length of list L
  L = [],           %   if L is an empty list
  S = 0.            %   its length S is zero
length(L,S):-       % otherwise
  L = [X|Xs],       % if L is a list [X|Xs]
  length(Xs,SXs),   % get the length SXs of its tail Xs
  S is 1 + SXs.     % add 1 to the length of its tail
```

- Simplified version:

```
length([],0).         % the empty list has length 0
length([_|Xs],S):-    % a non-empty list [_|Xs] has size S
  length(Xs,SXs),     % get the length SXs of its tail Xs
  S is 1 + SXs.       % add 1 to the length of its tail
```

Anonymous variable, used as "place holder".

In this context, we don't care what the value of the element is – we just want to count them!

SWI Prolog

UNIVERSITY OF ABERDEEN

Arithmetic - Summary

- Arithmetic is via "`is`" built-in.
- Some operators:
  - `+, -, *, /` (add, subtract, multiply, divide)
  - `//` (integer division)
  - `mod` (modulo)
- All variables on expression must have values, otherwise execution error.
- Because of this restriction, need to bear in mind the order in which *Prolog* proves/executes the body of a clause (or a query).

UNIVERSITY OF ABERDEEN

Failure-Driven Loops

- *Prolog* offers a built-in predicate **`fail`** which always fails:

```
?- fail.
false
```

- We can use this predicate to define a failure-driven loop, an alternative to recursion:

```
p(a).
p(b).
p(c).

loopFail:-      % loopFail succeeds if
  p(X),         %   we can prove p(X) and
  write(X),     %   write the value of X and
  nl,           %   skip a line.
  fail.         %   fail and BACKTRACK!
loopFail.       % if no (more) answers, stop
```

```
?- loopFail.
a
b
c
true
?-
```

UNIVERSITY OF ABERDEEN

Controlling Backtracking via Cuts (!)

- *Prolog*'s backtracking mechanism may, in some cases, lead to inefficiencies.
- We can control backtracking via the built-in "**!**", called "***cut***".
- The "**!**" is used as an ordinary predicate, in the body of the clause or query – it always succeeds!

- The "**!**" has a special property:
  - It causes the execution to commit to the solutions (proofs) of the goals to its left.
- Example:

```
p(A,B,C,D):- q(A), r(A,B), !, s(B,C),t(A,D).
```

UNIVERSITY OF ABERDEEN

Controlling Backtracking Using !

- Example:

```
p(a).
p(b).
p(c).
q(b,2).
q(c,3).
s(2).
s(3).
r(Y):- p(X),q(X,Y),!,s(Y).
```

```
?- r(Ans).
Ans = 2 ? ;
false
?-
```

Force backtrack…

$r(Y_1)$:- $p(X_1)$,$q(X_1,Y_1)$,!,$s(Y_1)$.
$\{X_1/a\}$ Fail & Backtrack…

$r(Y_1)$:- $p(X_1)$,$q(X_1,Y_1)$,!,$s(Y_1)$.
$\{X_1/b,Y_1/2\}$

$r(Y_1)$:- $p(X_1)$,$q(X_1,Y_1)$,!,$s(Y_1)$.
$\{X_1/b,Y_1/2\}$

$r(Y_1)$:- $p(X_1)$,$q(X_1,Y_1)$,!,$s(Y_1)$.
$\{X_1/b,Y_1/2\}$

$r(Y_1)$:- $p(X_1)$,$q(X_1,Y_1)$,!,$s(Y_1)$.
$\{X_1/b,Y_1/2\}$ No other proof for **$s(Y_1)$**

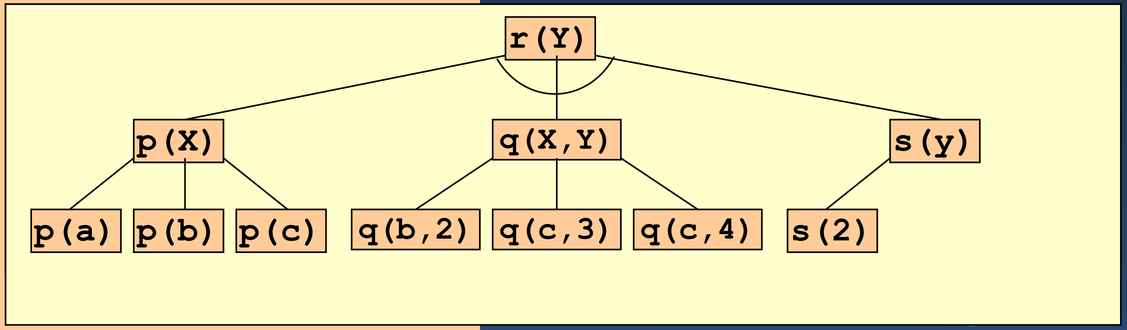
$r(Y_1)$:- $p(X_1)$,$q(X_1,Y_1)$,!,$s(Y_1)$.
$\{X_1/b,Y_1/2\}$ Cannot backtrack over "!"

UNIVERSITY OF ABERDEEN

Controlling Backtracking Using !

- Why "cut"?
  - It is a reference to the search space of a query…

```
p(a).
p(b).
p(c).
q(b,2).
q(c,3).
q(c,4).
s(2).
s(3).
s(4).
r(Y):-p(X),q(X,Y),!,s(Y).
```



with a cut (!)

UNIVERSITY OF ABERDEEN