CS2510
MODERN PROGRAMMING LANGUAGES

Logic Programming 3

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

Executing *Prolog* Programs

UNIVERSITY OF ABERDEEN

Executing *Prolog* Programs

- *Prolog* implements the resolution algorithm:

resolution(KB,Query):**boolean**
let Query be $\leftarrow C_1,\ldots,C_n$
for i = 1 to n **do**
  if there is a fact $A \leftarrow$ in KB such that $A = C_i$
  then true
  else if there is a rule $A \leftarrow B_1,\ldots,B_n$ in KB
    such that $A = C_i$
   then resolution(KB,$\leftarrow B_1,\ldots,B_n$)
   else false
if all C_i are **true then return true**
    else return false

- A fact or rule is chosen like this:
  – Scan program from top to bottom (left to right);
  – Pick out the first one.
- The test **`A = Ci`** (*unification*) is term matching.

• KB is a *Prolog* program
• Query is as defined
• How do we choose a fact or a clause?
• How do we test if $A = C_i$?

Unification

- Any term can be unified with itself.

```
weather(foggy) = weather(foggy)
```

- A variable can be unified with another variable.

```
X = Y
```

- A variable can be unified with (*instantiated to*) any *Prolog* term.

```
Topic = president(trump)
```

- Two different term structures can be unified if their constituents can be unified.

```
female(X) = female(jane)
mother(mary, X) = mother(Y, father(Z))
```

- A variable can even be unified with a term structure containing that same variable.

```
X = f(X)
```

SWI Prolog

UNIVERSITY OF ABERDEEN

Executing *Prolog* Programs

- Let's suppose we have edited the program:

```
rich(bill).
rich(arnie).
famous(bill).
friend(X):- rich(X),famous(X).
```

- Next we load it up in *Prolog* and type the query:

```
?- friend(F).
```

- The query means: "Who is a friend?"
- *Prolog* uses the resolution algorithm to find a solution to the query.

UNIVERSITY OF ABERDEEN

Executing *Prolog* Programs

- *Prolog* finds a rule whose head matches the query:

```
rich(bill).
rich(arnie).
famous(bill).
friend(X):- rich(X),famous(X).
```

- A fresh copy of the chosen rule is made:

```
friend(X1):- rich(X1),famous(X1).
```

(this prevents accidental name clashes)

- Because of the match, the variable **`F`** of the query is aliased (or bound) with the fresh variable **`X1`**.
- *Prolog* now has to prove the body of the clause it chose:

```
rich(X1), famous(X1).
```

Executing *Prolog* Programs

- *Prolog* tries to find a fact to match **`rich(X1)`**

```
rich(bill).
rich(arnie).
famous(bill).
friend(X):- rich(X),famous(X).
```

- The matching assigns **`bill`** to **`X1`**

- *Prolog* next tries to prove **`famous(X1)`** – however, it is now **`famous(bill)`**

```
rich(bill).
rich(arnie).
famous(bill).
friend(X):- rich(X),famous(X).
```

- *Prolog* finishes the execution and gives the value of the variable **`F`**

```
?- friend(F).
F = bill ?
```

At this point, *Prolog* offers (via the "?") the possibility of another execution, but choosing different rules/facts.

UNIVERSITY OF ABERDEEN

Backtracking

- What happens if we ask *Prolog*, in the previous example, for another answer?

```
?- friend(F).
F = bill ? ;
```

";" means: give me another answer (if there is one)

- *Prolog* tries to find another clause (fact/rule) for the last choice made.
  - Last choice: a fact that matched `famous(bill)`.
- *Prolog* tries to find another fact, but it doesn't find any.
- It then ***backtracks*** to the previous goal `rich(X1)`, undoing any assignments that had taken place.

Backtracking

- *Prolog* finds another fact that matches **`rich(X1)`**:

```
rich(bill).
rich(arnie).
famous(bill).
friend(X):- rich(X),famous(X).
```

- The matching assigns **`arnie`** to **`X1`**

- *Prolog* tries to prove **`famous(X1)`** – however, it is now **`famous(arnie)`**
- *Prolog* cannot find a fact/rule that matches it, and fails, issuing a "false" at the prompt:

```
?- friend(F).
F = bill ? ;
False
?-
```

Prolog is ready to take on another query

Backtracking

- When *Prolog* is trying to prove a query, it makes decisions on which fact/rule to use;
  - these choices may lead to failures.
- When *Prolog* fails a proof, it tries other facts/rules (if possible).
- If a fact/rule is not good, then *Prolog* chooses the one immediately after (top-down) the one it had chosen before.
- In order to return “false”, *Prolog* has to try all possible ways to prove a goal.
- When we type “;” at the prompt, we are forcing ***backtracking***.

Backtracking

- Another Example:
  - Same program, different order.

```
rich(arnie).
rich(bill).
famous(bill).
friend(X):- rich(X),famous(X).
```

```
?- friend(F).
```

- A similar execution takes place, however, when *Prolog* tries to prove **`rich(X1)`**, the first fact it finds is **`rich(arnie)`**
- The matching assigns **`arnie`** to **`X1`**
- *Prolog* tries to prove **`famous(X1)`** – however, it is now **`famous(arnie)`**
- *Prolog* cannot prove **`famous(arnie)`**

Backtracking

- *Prolog* backtracks to the last choice point and tries another clause.
- The last choice was **`rich(arnie)`** for **`rich(X1)`**, and it now chooses **`rich(bill)`**
- The rest of the proof is as before…

SWI Prolog

UNIVERSITY OF ABERDEEN

Programming in Prolog

- What kind of programming language is this?
  - NO built-in loops (do-while, do-until, for)
  - NO goto’s or if-then-else’s.

- There are, however, a number of built-ins:
  - Operators and arithmetic (`>, <, *, /, div, mod`)
  - I/O: `read, write, nl`
  - Graphics: windows, buttons, panels, etc;
  - Internet programming: fetch URLs, parse HTML, etc.
- If-then-else’s are offered as “syntactic sugar”:

```
p :- q -> r ; s.   % if q then r else s
```

UNIVERSITY OF ABERDEEN

Programming in Prolog

- The execution flow is given by the resolution proof procedure.
- As programmers, we take advantage of this.
- Iteration is achieved via recursion, i.e. a predicate defined in terms of itself.
- Variables are local to a rule/fact.
- There is no need to declare variables or their types.

Execution Example

(Work Through in Your Own Time)

UNIVERSITY OF ABERDEEN

Programming in Prolog

- An Example:

```
loop:-          % this loop predicate consists of
    read(L),    % reading a term from the standard Input (keyboard)
    L \= end,   % if input different from "end" then
    loop,       % carry on looping (until "end" is typed)
    write(L),   % then print the term typed (standard output – screen)
    nl.         % skip a line
loop.           % if input is "end", then this rule is used.
```

```
?- loop.
|: a.
|: b.
|: c.
|: end.
c
b
a

yes
?-
```

Can you follow the execution of this example?

UNIVERSITY OF ABERDEEN

Programming in Prolog

- ***Execution***:

```
loop:-
    read(L),
    L \= end,
    loop,
    write(L),
    nl.
loop.
```

```
?- loop.
|: a.
```

1. Prolog looks for clause (fact or rule) that matches `loop` – first one top-down is chosen;

2. Prolog creates fresh instance of the clause and tries to prove its body:
`read(L1),L1 \= end,loop,write(L1),nl.`

2.1 Prolog executes built-in, assigning a value to variable L_1:
`read(L1),L1 \= end,loop,write(L1),nl.`
`{L1/a}`

2.2 Prolog compares value of variable L_1:
`read(L1),L1 \= end,loop,write(L1),nl.`
`{L1/a}`

2.3 Prolog reaches recursive call – the same sequence of previous steps will be followed:
`read(L1),L1 \= end,loop,write(L1),nl.`
`{L1/a}`

UNIVERSITY OF ABERDEEN

Programming in Prolog

- ***1st Recursive Call***:

```
loop:-
    read(L),
    L \= end,
    loop,
    write(L),
    nl.
loop.
```

```
?- loop.
|: a.
|: b.
```

1. Prolog looks for clause (fact or rule) that matches `loop` – first one top-down is chosen;

2. Prolog creates fresh instance of the clause and tries to prove its body:
   `read(L2),L2 \= end,loop,write(L2),nl.`

2.1 Prolog executes built-in, assigning a value to variable L_2:
   `read(L2),L2 \= end,loop,write(L2),nl.`
   `{L2/b}`

2.2 Prolog compares value of variable L_2:
   `read(L2),L2 \= end,loop,write(L2),nl.`
   `{L2/b}`

2.3 Prolog reaches recursive call – the same sequence of previous steps will be followed:
   `read(L2),L2 \= end,loop,write(L2),nl.`
   `{L2/b}`

UNIVERSITY OF ABERDEEN

Programming in Prolog

- ***2nd Recursive Call*:**

```
loop:-
    read(L),
    L \= end,
    loop,
    write(L),
    nl.
loop.
```

```
?- loop.
|: a.
|: b.
|: c.
```

1. Prolog looks for clause (fact or rule) that matches `loop` – first one top-down is chosen;

2. Prolog creates fresh instance of the clause and tries to prove its body:
   read(L_3),L_3 \= end,loop,write(L_3),nl.

2.1 Prolog executes built-in, assigning a value to variable L_2:
   read(L_3),**L_3 \= end,loop,write(L_3),nl.**
   {L_3/c}

2.2 Prolog compares value of variable L_2:
   read(L_3),L_3 \= end,**loop,write(L_3),nl.**
   {L_3/c}

2.3 Prolog reaches recursive call – the same sequence of previous steps will be followed:
   read(L_3),L_3 \= end,**loop,write(L_3),nl.**
   {L_3/c}

Programming in Prolog

- *3rd (and final!) Recursive Call*:

```
loop:-
    read(L),
    L \= end,
    loop,
    write(L),
    nl.
loop.
```

```
?- loop.
|: a.
|: b.
|: c.
|: end.
```

1. Prolog looks for clause (fact or rule) that matches `loop` – first one top-down is chosen;

2. Prolog creates fresh instance of the clause and tries to prove its body:
   $\texttt{read(L}_4\texttt{),L}_4\texttt{ \\= end,loop,write(L}_4\texttt{),nl.}$

2.1 Prolog executes built-in, assigning a value to variable L_2:
   $\texttt{read(L}_4\texttt{),L}_4\texttt{ \\= end,loop,write(L}_4\texttt{),nl.}$
$\{L_4/\texttt{end}\}$

2.2 Prolog compares value of variable L_2:
   $\texttt{read(L}_4\texttt{),L}_4\texttt{ \\= end,loop,write(L}_4\texttt{),nl.}$
$\{L_4/\texttt{end}\}$

2.3 Prolog fails and backtracks, skips over `read` (no backtrack on built-ins) and tries another clause for `loop`:
   `loop.`

2.4 That's the end of this proof/run for `loop`. But this is not the end of the computation!! There are 3 pending computations to be finished!!

UNIVERSITY OF ABERDEEN

Programming in Prolog

- ***2nd Recursive Call (termination)***:

```
loop:-
    read(L),
    L \= end,
    loop,
    write(L),
    nl.
loop.
```

```
?- loop.
|: a.
|: b.
|: c.
|: end.
c
```

2.4 Prolog returns from recursive call and carries on left-to-right:

$\texttt{read}(L_3), L_3 \texttt{ \textbackslash= end}, \texttt{loop}, \texttt{write}(L_3), \texttt{nl}.$

$\{L_3/c\}$

2.5 Prolog finishes the clause, and "pops up" previous recursive call

$\texttt{read}(L_3), L_3 \texttt{ \textbackslash= end}, \texttt{loop}, \texttt{write}(L_3), \texttt{nl}.$

NB Variable L_3 and its value **c** cease to exist!!

Programming in Prolog

- *1st Recursive Call (termination)*:

```
loop:-
    read(L),
    L \= end,
    loop,
    write(L),
    nl.
loop.
```

```
?- loop.
|: a.
|: b.
|: c.
|: end.
c
b
```

2.4 Prolog returns from recursive call and carries on left-to-right:

read(L_2),L_2 \= end,loop,write(L_2),nl.

{L_2/b}

2.5 Prolog finishes the clause, and "pops up" previous recursive call

read(L_2),L_2 \= end,loop,write(L_2),nl.

NB Variable **L_2** and its value **b** cease to exist!!

Programming in Prolog

- *End of Execution*:

```
loop:-
    read(L),
    L \= end,
    loop,
    write(L),
    nl.
loop.
```

```
?- loop.
|: a.
|: b.
|: c.
|: end.
c
b
a

yes
?-
```

2.4 Prolog returns from recursive call and carries on left-to-right:

`read(L1),L1 \= end,loop,`**`write(L1),nl.`**

{L1/a}

2.5 Prolog finishes the clause, and terminates execution!

`read(L1),L1 \= end,loop,write(L1),`**`nl.`**

NB Variable **L_1** and its value **a** cease to exist!!

UNIVERSITY OF ABERDEEN

Nonmonotonic logic

- Standard logic is ***monotonic***.
  - once you prove something is true, it is true forever.
- *Prolog* uses ***nonmonotonic*** logic
  - facts and rules can be changed at any time;
  - such facts and rules are said to be *dynamic*.

- **`assert(...)`**
  - adds a fact or rule
- **`retract(...)`**
  - removes a fact or rule

- `assert` and **`retract`** are said to be ***extralogical*** predicates.
- `assert` and **`retract`** are not undone by backtracking.

Examples of `assert` and `retract`

```
assert(man(plato)).

assert((loves(owen,X) :- female(X), rich(X))).
```

Asserts the fact `man(plato)` into the KB.

```
retract(man(plato)).

retract((loves(owen,X) :- female(X), rich(X))).
```

Retracts the rule `loves(owen, X)` from the KB.

- CARE using `assert` and `retract`!
- Changing the *Prolog* KB at run-time can make programs unpredictable, hard to debug.

- Notice that we use double parentheses for rules:
  - this is to avoid a minor syntax problem
  - `assert(foo :- bar,  baz).`
  - How many arguments did we give to assert?

UNIVERSITY OF ABERDEEN

Lists in Prolog

- Data structure for non-numeric programming.
- A sequence of any number of items.
- The items can be any *Prolog* term (including nested lists!).
- Useful when we don't know in advance how many items we are going to have.
- Flexible and generic.
- In *Prolog*, a list is represented as a sequence of terms, separated by commas and within **[…]**.

```
[123, hello, Var1, p(a,X), [1,2,3]]
```

UNIVERSITY OF ABERDEEN

Lists in Prolog

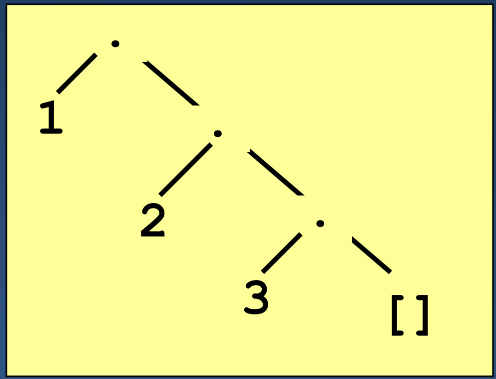
- Lists are ordinary *Prolog* **terms**.
- The notation `[…]` is just a shorthand.

`[1,2,3]` ≡ `.(1,.(2,.(3,[])))`

- Lists are thus represented internally as trees
  - The last element is always the empty list `[ ]`

```
?- List = [1,2,3].
List = [1,2,3]
true

?-
```



Lists – [H|T]

- *Prolog* provides a notation for manipulating lists.
- The notation is **`[H|T]`**
  - Shortand for **`.(H,T)`**
  - ***`H`*** stands for the first element of the list.
  - ***`T`*** stands for the remaining elements of the list and must be a list.
- We can build and decompose lists with **`[H|T]`**:

```
?- List = [1,2,3], List = [A|B].
A = 1,
B = [2,3],
List = [1,2,3]

?- List = [A|B], A = 1, B = [2,3].
A = 1,
B = [2,3],
List = [1,2,3]
```

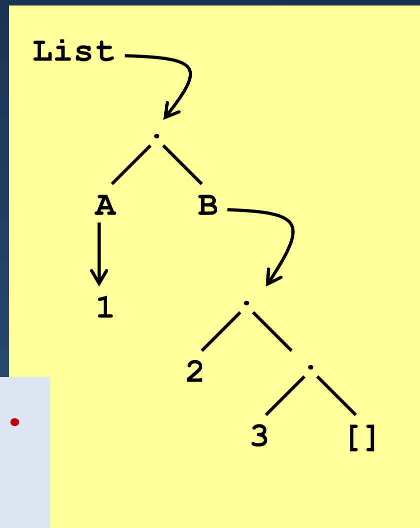
Lists – [H|T]

- The `[H|T]` notation allows for variations:

  `[E1,E2|T]`
  (a list with at least 2 elements)

  `[1|[2,3|T]]`
  (a list [1,2,3|T])

- `[H|T]` is also a term, hence a tree:




```
?- List = [A|B], A = 1, B = [2,3].
A = 1,
B = [2,3],
List = [1,2,3]
```

Basic Operations on Lists

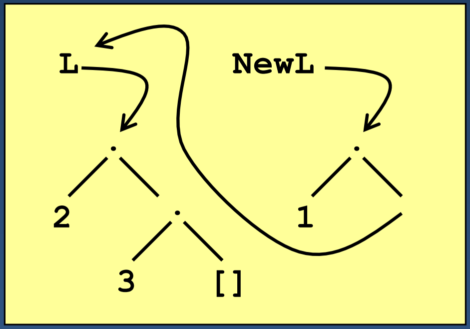
- We can *access* elements from the front of a list.

```
?- [1,2,3] = [X|Y].
X = 1,
Y = [2, 3]

?- [jan,feb,mar] = [M,N|R].
M = jan,
N = feb,
R = [mar]
```

- We can *add* elements to the front of a list:

```
?- L = [2,3], NewL = [1|L].
L = [2, 3],
NewL = [1, 2, 3]
```



UNIVERSITY OF ABERDEEN