Processadors de Llenguatge II

Logic Programming
Sebesta Ch.16
Course webpage

Rafael Ramirez
Dep Tecnologia
Universitat Pompeu Fabra
Programming languages

A Program (in some language)
= \textit{Data} + \text{Operations} + \textit{Control}
<table>
<thead>
<tr>
<th>Data</th>
<th>+ Operations</th>
<th>+ Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different Paradigms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Discuss issues across paradigms.
### Paradigms

We have to go horizontally.

<table>
<thead>
<tr>
<th>Different Paradigms</th>
<th>Data + Operations</th>
<th>+ Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The logic paradigm

- **Programs** are Predicate Calculus **Formulas**
- **Execution** is reduced to **proving** a predicate by deduction by modus ponens
Prolog is an approximation of Logic Programming.

Execution Order:
- Prolog specifies the order in which clauses are chosen.
- Specific order of resolution of the goals in the body of a clause.
- This provides backtracking to simulate the non-determinism.
Meta-logical features:
- Prolog provides facilities for enhancing the expressiveness of the logic programming model.
- E.g. Var, !, functor, ...

Restricted arithmetic expressions
- an operand of an arithmetic operator cannot be an uninstantiated logical variable
Prolog

- In prolog, think **RELATIONS**.
- This is the paradigm shift.
- Relations are more general than functions.
A **predicate** is an **atom** followed by an argument list of terms. Predicate names model **relations**.
Prolog Syntax

\[

term \::= \text{number} \mid \text{atom} \mid \text{variable} \\
\mid \text{function} \ ( \ \text{term} \ \{, \ \text{term} \} \ )
\]

\[

atom \::= \text{smallalpha}\{\text{alphanum}\}
\]

\[

function \::= \text{smallalpha}\{\text{alphanum}\}
\]

\[

variable \::= \text{largealpha}\{\text{alphanum}\}
\]

\[

\text{smallalpha} \::= \text{a} \mid \text{b} \mid \ldots \mid \text{z}
\]

\[

\text{largealpha} \::= \text{A} \mid \text{B} \mid \ldots \mid \text{Z}
\]

\[

\text{number} \::= \text{digit}\{\text{digit}\}
\]

\[

\text{digit} \::= 0 \mid 1 \mid \ldots \mid 9
\]

\[

\text{alphanum} \::= \{\text{smallalpha} \mid \text{largealpha} \mid \text{digit}\}
\]

Atoms **must** begin with a small letter.
Variables **must** begin with a capital letter.
A predicate by itself, is known as a fact in prolog.
This is how we represent / model knowledge.
Variables in facts are UNIVERSALLY QUANTIFIED.
Example:

- link(fortran,algol60).
- link(c,cplusplus).
- link(algol60,cpl).
- link(algol60,simula67).
- link(cpl,bcpl).
- link(simula67,cplusplus).
- link(bcpl,c).
- link(simula67,smalltalk80).
**Rules**

- Rules are of the form:
  
  \[
  \langle \text{pred}_0 \rangle \ :- \ \langle \text{pred}_1 \rangle \ , \ \langle \text{pred}_2 \rangle \ , \ \ldots \ , \ \langle \text{pred}_n \rangle .
  \]

- They denote the equivalent logical meaning of
  
  \[
  (\langle \text{pred}_1 \rangle \ \land \ \langle \text{pred}_2 \rangle \ \land \ldots \land \ \langle \text{pred}_n \rangle) \ \rightarrow \ \langle \text{pred}_0 \rangle
  \]

- Variables ...
  
  - in the **HEAD OF THE RULE** are **UNIVERSALLY QUANTIFIED**.
  - in the **BODY OF THE RULE** which are not ‘bound’ to those in the head, are **EXISTENTIALLY QUANTIFIED**.
Example

path(X,X).

path(X,Y) :- link(X,Z), path(Z,Y).

Fact states that all languages have a path to themselves: ∀X, path(X,X)

Rule: Any language X has a path to any language Y, if there exists some Z such that X is directly linked to Z, and that Z has a path to Y.
Queries

\[
\begin{align*}
\text{<query>} & \ ::= \text{<pred>} \ {,} \ \text{<pred>}. \\
\text{<pred>} & \ ::= \text{<atom>} \ (\ \text{<term>} \ {,} \ \text{<term>})
\end{align*}
\]

• A query is a list of predicates separated by ‘,‘.
• Variables in a query are existentially quantified.
• A solution to a query is a binding of variables (in the query) to values that makes the query true.
Queries

- Example:
  \[
  \text{link(algo60,L), link(L,M).}
  \]
  ...is equivalent to predicate calculus:
  \[
  \exists L, \exists M, (\text{link(algo60,L)} \land \text{link(L,M)})
  \]

  “Does there exist some values for L and M such that algo60 is linked to L and L is linked to M?”
Logic programming execution model

**Input:** A logic program $P$ and a goal $G$

**Output:** Binding $B$ of variables in $G$ or failure.

**Algorithm:**

Initialize the resolvent to be $G$, the input goal.
Initialize $U$ to empty.

While the resolvent is not empty do

- Choose a goal $A$ from the resolvent and a (renamed) clause $A' \leftarrow B_1, B_2, ..., B_n$ from $P$ such that $A$ and $A'$ unify with mgu $U'$ (exit if no such goal and clause exist).
- Remove $A$ from the resolvent and add $B_1, B_2, ..., B_n$ to the resolvent.
- Apply $U'$ to the resolvent
- set $U = UU'$ (composition of $U$ and $U'$)

end while

If the resolvent is empty then output $U$, else output failure.
Prolog execution model

- How does prolog evaluate the query?
  
  \[ \text{path(algo160,L)}. \]

- Model of Computation:
  - Deduction via Modus Ponens rule
  - Unification
  - Backtracking;
The leftmost goal of the resolvent is chosen,

The non-deterministic choice of a clause that unifies with the goal is replaced by a sequential search, determined by the order in which the clauses were written.

If no clause can be unified with the popped goal, then the computation is unwound to the last choice of clause made and the next unifiable clause is found. This is backtracking.
Execution model

- Suppose we have a query \( s(a) \) and a program

\[
\begin{align*}
(a) & \quad s(X) :- q(Y), r(X, Y) \\
(b) & \quad q(X) :- p(X) \\
(c) & \quad p(b) \\
(d) & \quad r(a, b)
\end{align*}
\]
Execution model

- Now query $m(a)$ and program

\[\begin{align*}
(a) & \quad m(X) \ :- s(X), n(b) \\
(b) & \quad m(X) \ :- s(X), n(X) \\
(c) & \quad s(X) \ :- l(X) \\
(d) & \quad s(X) \ :- q(Y), r(X, Y) \\
(e) & \quad q(X) \ :- p(X) \\
(f) & \quad p(b) \\
(g) & \quad r(a, b) \\
(h) & \quad l(c) \\
(i) & \quad n(a)
\end{align*}\]
Query evaluation

- Query evaluation involves the use of:
  - Modus Ponens rule.
  - Unification
  - Backtracking
Modus ponens

Modus Ponens rule:

\[
\begin{array}{c}
P \\
\rightarrow \\
P \rightarrow Q \\
\hline
Q
\end{array}
\]

P and (P \rightarrow Q) therefore Q.

Prolog works backwards:

- If I want to prove Q and
- I know that P \rightarrow Q
- All I need to do is to show that P is true.

\[
\text{path}(X,Y) := \text{link}(X,Z) \land \text{path}(Z,Y).
\]

\[
\begin{array}{c}
Q \\
\hline
P
\end{array}
\]
Unification

- In matching the Goal
  \[
  \text{path(\text{algol60},L)}
  \]
  with the Head of Rule,
  \[
  \text{path(X,Y) :- link(X,Z) , path(Z,Y)}.
  \]
  unification is performed.

- Unification is \textbf{NOT} assignment.
- Think of unification as an association.
Unification

- The ‘=’ is NOT assignment. Neither is it a boolean comparison.
- ‘LHS = RHS’ means LHS is unified with RHS.
- Unification does a structural comparison of the two terms. And so:
  - Unification is a reflexive, symmetric and transitive relation.
  - Unification does NOT involve computation.
Unification

- Reflexive
  
  ?- X = X.
  X = _G190
  Yes

  ?- 3 = 3.
  Yes

- Symetric

  ?- X = 3.
  X = 3 Yes

  ?- 3 = X.
  X = 3 Yes
Unification

- Transitive
  
  `- X = Y, Y = Z, X = 3.
    X = 3    Y = 3    Z = 3    Yes

  `- X = Y, Y = Z, Z = 3.
    X = 3    Y = 3    Z = 3    Yes

- Unification does NOT involve computation:
  
  `- X = 3 + 1.
    X = 3+1    Yes

  `- 3+1 = X.
    X = 3+1    Yes

  `- 2+2 = 3+1.
    No
Unification is structural comparison:

?- sit(john,table) = sit(john,X).
X = table
Yes

?- sit(Y,table) = sit(john,X).
Y = john
X = table
Yes

?- love(sister(X,john),Y) = love(Z,brother(3,peter)).
X = _G457
Y = brother(3,peter)
Z = sister(_G457,john)
Yes
Unification

Unification

Rule 1: $\text{path}(_1, _1)$

\{_1 = \text{algol60}, \_1 = L\}\n
Rule 2: $\text{path}(_1, _2) \leftarrow \text{link}(_1, _3), \text{path}(_3, _2)$

\{_1 = \text{algol60}, \_2 = L\}\n
$L = \text{algol60}$

link(\text{algol60}, _3), \text{path}(_3, L)$
A unification algorithm

Input: Two terms T1 and T2 to be unified
Output: U the most general unifier of T1 and T2, or failure.

Algorithm:
Initialize the substitution U to be empty, the stack to contain the equation T1 = T2, and failure to false.

While stack not empty and no failure do
  pop X = Y from the stack
  case
    X is a variable that does not occur in Y:
      substitute Y for X in the stack and in U.
      add X = Y to U.
    Y is a variable that does not occur in X:
      substitute X for Y in the stack and in U.
      add Y = X to U.
    X and Y are identical constants or variables: continue
    X is f(X1, ..., Xn) and Y is f(Y1, ..., Yn) for some functor f and n>0:
      push Xi = Yi, for all i(1..n), on the stack.
    otherwise:
      failure := true;
  end while
if failure, then output failure else output U.
Recall that variables in a prolog query are existentially quantified.

There may exist more than one solution to the query.

Backtracking is used to search for other possible solutions which satisfy the query.

And to try an alternative path in the search tree when a branch fails.
Backtracking
Termination and Order of Evaluation

Consider the following program:

parent(tom,mary).
parent(mary,sue).
parent(mary,john).
ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).

And the query:

ancestor(A,B)
Computation

- It (1) evaluates the arithmetic expression on the right hand side, before (2) unifying it to the term.

- Evaluate, and then unify.

- NOT evaluate and assign.
  (No destructive assignment in prolog)
Examples

?- X is 2+1.
X = 3    Yes

?- 2+1 is X.
ERROR: Arguments are not sufficiently instantiated

?- 3 is 2+1.
Yes

?- 2+1 is 3.
No
No data types!!!
Every data is a term!!! (This is the only type)
All terms are regarded as trees.
For example, a term $p(a_1, ..., a_n)$ is represented as a tree:
Structures and Types

- So terms $p(a_1, \ldots, a_n)$ can be viewed as a constructor $p$ with branches $a_1$ to $a_n$.

- How about the numbers and atoms?

- They can be viewed as a tree constructor with **no** branches!

- (That is why unification does a structural comparison between two trees!)
Example

Here’s a ‘tree’ constructed using relation mycons, and atom mynil.

?- X = mycons(1,mycons(2,mynil)).
X = mycons(1, mycons(2, mynil))   Yes

Now, you may think that there is some type restriction on the things stored? No there isn’t!

Prolog just has terms. There is no type enforcement. So the following is also allowed.

?- X = mycons(mycons(2.2,mynil),1).
X = mycons(mycons(2.2, mynil), 1) Yes
Example

- Get used to no types

?- X = a(a,a,a,a,a).
X = a(a,a,a,a,a).
Yes

?- X = a(1,b(f,g),2.7,a,c(d,c),e(e(e(1,e),f)))).
X = a(1,b(f,g),2.7,a,c(d,c),e(e(e(1,e),f)))).
Yes
Structures and Types

- There are some default structures available for your use:
  - Lists: items within [ and ]
  - Tuples: items within ( and )

- List head and tail are unified (pattern-matched) through ‘|’

?- X = [1,2,3] , Y = [10,11 | X] , Z = [X | X].
X = [1, 2, 3]
Y = [10, 11, 1, 2, 3]
Z = [[1, 2, 3], 1, 2, 3]
Yes
Lists

You can define a rule to add up the elements of a list

\[
\text{addup}([],0).
\]
\[
\text{addup}([X|Xs],R) :- \text{addup}(Xs,Tmp), R \text{ is } Tmp+X.
\]

You query using:

\[
?- \text{addup}([100, 200, 300],X).
\]
\[
X = 600
\]
Yes.
Lists

Trace the unification process using a search tree.

addup([],0).
addup([X|Xs],R) :- addup(Xs,Tmp), R is Tmp + X.

addup([100, 200, 300],X)

Rule 1: Fail

Rule 2: Matched {_1=100, _2=[200,300], _3=X}

addup([200,300],_4) , X is _4 + 100

Rule 1: Fail

Rule 2: Matched {_5=200, _6=[300], _7=_4}

addup([300],_8) , _4 is _8+200, X is _4+100
Lists

addup([],0).
addup([X|Xs],R) :- addup(Xs,Tmp), R is Tmp+X.

Evaluate and Unify: Success: _4 = 500

Evaluate and Unify: Success: x = 600

addup([300],_8), _4 is _8+200, X is _4+100

Rule 1: Fail
Rule 2: Matched

Evaluate and Unify: Success: _4 = 500

addup([_9|_10],_11) :-
    addup(_10,_12), _11 is _12 + _9.
    {_9=300, _10=[], _11=_8}

2nd sub goal: Evaluate and Unify: Success!

_8 = 300
Return
Lists

- Appending two lists

```prolog
app([],YS,YS).
app([X|XS],YS,ZS) :- app(XS,YS,TMP), ZS=[X|TMP].
```

- You query using:

```prolog
?- app([1,2],[3,4,5],OUT).
OUT = [1,2,3,4,5]
Yes.
```

- Same as Concat/3
Tuples

- Constructed with n-elements in (..) brackets.
- Unification can be applied to them.

?- X = (1,2,(3,4,5),6).
X = 1, 2, (3, 4, 5), 6
Yes
Tuples

- Tuples are different from lists. Although their behaviour in unification may give you an otherwise impression.

  - `?- (1,2)=[1,2].`
  - No

  - `?- [1,2,3,4] = (X,Y).`
  - No
Closed world assumption

- Prolog has no knowledge of the world other than its own ‘database’.
- This means the prolog inference engine (1) depends on, and (2) is limited to, the accuracy of the rules and facts that you give to it.
As a consequence, Prolog is able to prove that something is true, but **not** able to **prove** that something is **false**.

Proven True in the real world | Proven False in the real world
---|---
Proven True in Prolog | “Fail” to find a solution
Closed world assumption

- So prolog is **not** a true/false system.
- It is a true/"I-don’t-know" system. i.e. a true/fail system.

- Do not confuse logical falsehood (in other languages) with prolog’s proof failure.
- Logical falsehood says:
  - “I can prove that it is false.”
- Prolog’s “no” (proof failure) says:
  - “I don’t know... I can’t prove that it is true, (neither can I prove that it is false). So I’ll assume that it is false.”
Negation in Logic Programming

- Logic Programs describe what is true. Untrue facts are simply omitted. *(closed world assumption)*

- Negation can be added to logic programming by introducing a `not` relation (that is only a partial form of negation in logic).

- A goal `not G` is a consequence of a program P if G is not a consequence of P.

- Negation is characterized by *failure*, that is `not G` succeeds if G fails.
Negation

- Prolog allows negations via the “not” operator.

Example:

```prolog
parent(bill,jake).
parent(bill,shelly).
sibling(X,Y) :- parent(M,X) , parent(M,Y).
```

If we query:

```prolog
sibling(X,Y).
```

Prolog will return

- X = jake   Y = jake
- X = jake   Y = shelly
- X = shelly Y = jake
Negation

- Fix the problem by using ‘not’:

parent(bill,jake).
parent(bill,shelly).
sibling(X,Y) :- parent(M,X) , parent(M,Y)
                , not(X=Y).

?- sibling(X,Y).
X = jake         Y = shelly ;
X = shelly       Y = jake ;
No
Negation

BUT because of the true/fail system, negations pose a problem.

Prolog’s “not” is different from the logical “not”!!!
Example 1 of a negation problem:

parent(bill,jake).
parent(bill,shelly).
sibling(X,Y) :- not(X=Y),
               parent(M,X),
               parent(M,Y).

Consider the following queries

?- sibling(X,Y).
   No
Negations – problem

- Example 2 of a negation problem:
  home(X) :- not(out(X)).
  out(sue)

- Consider the following queries

  ?- home(john).
  yes

  ?- home(X).
  no
Negations – problem

- Example 3 of a negation problem:
  
  \[
  \begin{align*}
  \text{even}(0). \\
  \text{even}(s(s(X))) & :\; \text{even}(X). \\
  \text{odd}(X) & :\; \text{not}(\text{even}(X)).
  \end{align*}
  \]

- Consider the following queries

  \[
  \begin{align*}
  \text{?- odd}(s(0)). \\
  \text{yes}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{?- odd}(X). \\
  \text{no}
  \end{align*}
  \]
Negations – problem

- How to avoid the problem?

- There are two kinds of goals.
  - **Test Goal** – all variables have been instantiated.
    - Eg. X = 3, Y = 4, not(X=Y).
  - **Finding Goal** – uninstantiated variables exist in the goal.
    - Eg. not(X=Y), X = 3, Y = 4.

- Use “not(G)” as a **test goal** and not as a finding goal. Meaning that: make sure that the variables in ‘G’ are already instantiated before you ‘not’ the ‘G’.
Negations – Implementation

- The negation is implemented as follows
  \[
  \text{not}(G) :- G, !, \text{fail}.
  \]
  \[
  \text{not}(G).
  \]
- The “!” is the cut operator
- The “fail” forces backtracking at that point.
- People call this “negation-as-failure”, meaning to say “negation implemented as failure”.
Control

- Start with a query as the current goal.
- Let the current goal be made up of sub-goals $G_1, \ldots, G_m$, $m > 0$.
- For each $G_j$ (i.e. try to satisfy each goal from left to right):
  - Let $R_1, \ldots, R_n$, $n > 0$ be all the rules applicable to this sub-goal.
  - For each $R_k$ (i.e. try to apply the rules from top to bottom):
    - If rule $R_k$ applies to $G_j$, then
      - Unify the variables,
      - form a new current goal and try to fulfill them again from left to right.
    - Backtrack when can’t find any more rules to apply.
  - If all sub-goals satisfied, then success, else fail.
Search tree depends on how you order your sub-goals and rules.

Different orderings...

- may cause the results to come out in a different order, or
- may cause non-termination of searches,
Control

link(simula67,cplusplus).
link(simula67,smalltalk80).

path(X,X).
path(X,Y) :- link(X,Z), path(Z,Y).

?- path(simula67,T).
T = simula67 ;
T = cplusplus ;
T = smalltalk80 ;
no

Rule Re-ordering

path(X,Y) :- link(X,Z), path(Z,Y).
path(X,X).

?- path(simula67,T).
T = cplusplus ;
T = smalltalk80 ;
T = simula67 ;
no

Reordering may cause results to come out in a different order
Control

link(simula67,cplusplus).
link(simula67,smalltalk80).

path(X,X).
path(X,Y) :- link(X,Z) , path(Z,Y).

?- path(simula67,T).
T = simula67 ;
T = cplusplus ;
T = smalltalk80 ;
no

Sub-goal Re-ordering

path(X,X).
path(X,Y) :- path(Z,Y) , link(X,Z).

?- path(simula67,T).
T = simula67 ;
T = cplusplus ;
T = smalltalk80 ;
Control

link(simula67,cplusplus).
link(simula67,smalltalk80).

\begin{align*}
\text{path}(X,X). \\
\text{path}(X,Y) & :\! :\! \text{link}(X,Z) \land \text{path}(Z,Y).
\end{align*}

?- \text{path}(\text{simula67},T).
T = \text{simula67} ;
T = \text{cplusplus} ;
T = \text{smalltalk80} ;
no

\begin{align*}
\text{path}(X,X). \\
\text{path}(X,Y) & :\! :\! \text{path}(Z,Y) \land \text{link}(X,Z).
\end{align*}

?- \text{path}(\text{simula67},T).
T = \text{simula67} ;
T = \text{cplusplus} ;
T = \text{smalltalk80} ;
Stack Overflow

\textbf{Sub-goal Re-ordering}

\textbf{Reordering may cause non-termination}
Control

link(simula67,cplusplus).
link(simula67,smalltalk80).

path(X,X).
path(X,Y) :- link(X,Z), path(Z,Y).

?- path(simula67,T).
T = simula67 ;
T = cplusplus ;
T = smalltalk80 ;
no

path(X,Y) :- path(Z,Y), link(X,Z).
path(X,X).

?- path(simula67,T).
Stack Overflow...

Worse still, non-termination occurs immediately
Control

- Re-ordering of sub-goals and rules affect the search of the tree... and so it may affect the results.

- Be careful how you write your rules. Take into account how the search will be carried out.

- You acquire experience in the search by tracing through some prolog queries.
One program – many behaviours

- Because **prolog deals with relations**, there are different ways in querying: each position in the term can be queried (**output**, ↑) or supplied with data (**input**, ↓).

- This is **unlike functions** (where all the arguments are the inputs, and a single result – output, is returned).
One program – many behaviours

Example 1:

\[
\begin{align*}
\text{link} & (\text{fortran}, \text{algol60}). \\
\ldots \\
\text{path} & (X, X). \\
\text{path} (X, Y) & :\!- \text{ link} (X, Z), \text{ path} (Z, Y).
\end{align*}
\]

Query of the form \(\text{path}(\downarrow, \downarrow)\):

Example:

\[\text{path(algol60, fortran)}.\]

“Is there a path from algol60 to fortran? Yes? No?”
One program – many behaviours

- Query of the form path(\(\downarrow, \uparrow\)):  
  Example:  
  \[
  \text{path(algol60,X)}.
  \]
  “Give me languages which have a path from algol60.”

- Query of the form path(\(\uparrow, \downarrow\)):  
  Example:  
  \[
  \text{path(X,algol60)}.
  \]
  “Give me languages which have a path to algol60.”
One program – many behaviours

- Query of the form path($\uparrow, \uparrow$):
  
  Example:
  
  \[
  \text{path}(X,Y). 
  \]

  “Give me two languages which have a path from one to the other.”

- All the different forms: 1-program, many-behaviours!
  
  - \(\text{path}(\downarrow, \downarrow)\) : Is there a path from ___ to ___?
  - \(\text{path}(\downarrow, \uparrow)\) : Languages which have a path from ___?
  - \(\text{path}(\uparrow, \downarrow)\) : Languages which have a path to ___?
  - \(\text{path}(\uparrow, \uparrow)\) : What are the possible paths?
One program – many behaviours

- In theory, for a term with $n$ argument positions, there are $2^n$ ways to query, each position can be a variable or some data.

- In practice, not all ways may be able to be used, as some may give non-termination.
  - For the previous example, all the variations can be used and will terminate. (Trace through it to understand why)
  - For the next example, you’ll see that not all variations can terminate…
One program – many behaviours

Example 2:

\[
\text{app}([\ ], Ys, Ys).
\]
\[
\text{app}([X|Xs], Ys, [X|Zs]) \leftarrow \text{app}(Xs, Ys, Zs).
\]

a. Query of the form \text{app}(\downarrow, \downarrow, \downarrow):

Example:

\[
\text{app}([1,2],[a,b],[1,a,2,b]).
\]

“Is the third list a result of appending the first to the second? Yes/No?”
Example 2:

\[
\text{app}([], Ys, Ys). \\
\text{app}([X|Xs], Ys, [X|Zs]) :- \text{app}(Xs, Ys, Zs).
\]

b. Query of the form \text{app}(\downarrow, \downarrow, \uparrow):

Example:

\[
\text{app}([1,2],[a,b],Z).
\]

“Append the first to the second list. What is the result?” (This is the append as we know it)
One program – many behaviours

Example 2:

\[
\begin{align*}
\text{app}([], Ys, Ys). \\
\text{app}([X|Xs], Ys, [X|Zs]) & : \text{app}(Xs, Ys, Zs).
\end{align*}
\]

c. Query of the form \(\downarrow, \uparrow, \downarrow\):

Example:

\[
\text{app}([1,2], X, [1,2,a,b]).
\]

“Is there something that I append behind [1,2] to get [1,2,a,b]? Is it possible? If so, what do I append”

Is [1,2] a \textbf{PREFIX} of [1,2,a,b]?
One program – many behaviours

Example 2:

\[
\begin{align*}
\text{app}([], Ys, Ys). \\
\text{app}([X|Xs], Ys, [X|Zs]) & :\ = \text{app}(Xs, Ys, Zs).
\end{align*}
\]

d.Query of the form \text{app}(\uparrow, \downarrow, \downarrow): 

Example:

\[
\text{app}(X, [b], [1, 2, a, b]).
\]

“Is there something that I append in front of [b] to get [1,2,a,b]? Is it possible? If so, what is that list?”

Is [b] a \textbf{SUFFIX} of [1,2,a,b]?
One program – many behaviours

Example 2:

\[
\begin{align*}
\text{app}([], Ys, Ys). \\
\text{app}([X|Xs], Ys, [X|Zs]) &: \text{ app}(Xs, Ys, Zs).
\end{align*}
\]

e.Query of the form \(\text{app}(\uparrow, \uparrow, \downarrow)\):

Example:

\[
\text{app}(X, Y, [1, 2, a, b]).
\]

“What 2 lists do I append together to get [1,2,a,b]?”

Find different ways to SPLIT the list. Many results, but will terminate.
One program – many behaviours

Example 2:

\[
\text{app}([], Ys, Ys). \\
\text{app}([X|Xs], Ys, [X|Zs]) :- \text{app}(Xs, Ys, Zs).
\]

f. Query of the form \(\text{app}(\uparrow, \downarrow, \uparrow)\):

Example:

\[
\text{app}(X, [1, 2, a, b], Y).
\]

“All possible lists ending with [1,2,a,b]”

Infinite results, and will NOT terminate (but still within your control through keying in the semi-colon).
One program – many behaviours

Example 2:

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

All the different forms: 1-program, many-tasks!

- `app(↓,↓,↓)`: Check whether appended correctly.
- `app(↓,↓,↑)`: ‘Normal’ append with 2 list input and result.
- `app(↓,↑,↓)`: Checks for Prefixes (returns the suffix)
- `app(↑,↓,↓)`: Checks for Suffixes (returns the prefix)
- `app(↑,↑,↓)`: List splitting

Due to many usages in different contexts, the name “app” may become misleading, because app will only ‘append’ the way we know it (as a function) when it is used ↓,↓,↑.
One program – many behaviours

Example 2:
\[
\text{app([],Ys,Ys).} \\
\text{app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).}
\]

Example: Different Context, different uses...

\[
p(X, Y, Z) :- \text{app}(X,_,Y), \text{app}(X,_,Z).
\]

What does \( p(↑, \downarrow, \downarrow) \) try to do?... Will it terminate?
Non-Determinism, Backtracking

- A programming language is **deterministic** if at any point in the execution, there is always exactly one step to in which it proceeds.

- A prolog predicate (seen as a procedure) may have **multiple clauses** (definitions).

- This clauses (definitions) **may not be mutually exclusive** (i.e. taking one clause will exclude the rest).

- Furthermore, whether a definition is mutually exclusive, **depends** on the way the query is formed.

- When the set of clauses (wrt query) are **not mutually exclusive**, **backtracking** is normally involved in the search.
A ‘cut’ (denoted with ‘!’) cuts out a part of a Prolog search tree.

\[ n(\ldots) :- \ldots \]
\[ \ldots \]
\[ n(\ldots) :- \ldots, \text{p}(\ldots), \ldots \]
\[ \ldots \]
\[ n(\ldots) :- \ldots \]

\[ \text{p}(\ldots) :- \ldots \]
\[ \ldots \]
\[ \text{p}(\ldots) :- \ldots, \text{q}(\ldots), !, \text{r}(\ldots), \ldots \]
\[ \ldots \]
\[ \text{p}(\ldots) :- \ldots \]

Q: What happens when we backtrack past a cut?
A ‘cut’ (denoted with ‘!’) cuts out a part of a prolog search tree.

\[
\begin{align*}
n(\ldots) & : - \ldots \\
& \ldots \\
n(\ldots) & : - \ldots, p(\ldots), \ldots \\
& \ldots \quad \text{Continue to backtrack from this point} \\
n(\ldots) & : - \ldots
\end{align*}
\]

\[
\begin{align*}
p(\ldots) & : - \ldots \\
& \ldots \\
p(\ldots) & : - \ldots, q(\ldots),!, r(\ldots), \ldots \\
& \ldots \\
p(\ldots) & : - \ldots
\end{align*}
\]

When ‘r’ fails, ‘p’ will also fail
Cuts

Example:

\[ \begin{align*}
  a(1) & : - b(1). \\
  a(2) & : - e(2). \\
  b(X) & : - c(X). \\
  b(X) & : - d(X). \\
  c(X). \\
  d(X). \\
  e(X). \\
\end{align*} \]

?- a(X).

Output:
\[ \begin{align*}
  & X = 1; \\
  & X = 1; \\
  & X = 2; \\
  & \text{No} \\
\end{align*} \]
Cuts

Example:

\[ a(1) :- b(1). \]
\[ a(2) :- e(2). \]
\[ b(X) :- !,c(X). \]
\[ b(X) :- d(X). \]
\[ c(X). \]
\[ d(X). \]
\[ e(X). \]

?- a(X).
X = 1 ;
X = 1 ;
X = 2 ;
No

Output:
X=1
Backtrack

Output:
X=1
Backtrack

Output:
X=2

\{X=1\} \quad \{X=2\}

! , c(1) \quad d(1)

Output: X=1
Backtrack

Output: X=2
Backtrack
Cuts

Example:

\[
\begin{align*}
a(1) & : - b(1). \\
a(2) & : - e(2). \\
b(X) & : - !, c(X). \\
b(X) & : - d(X). \\
c(X). \\
d(X). \\
e(X). \\
\end{align*}
\]

?- a(X).

Output: 
\[
\begin{align*}
X & = 1 \\
X & = 1; \\
X & = 2; \\
\text{No}
\end{align*}
\]

Diagram:

```
  a(X)  
 / \   
/   \  
{X=1} b(1) {X=2}
    /   
   /    
  e(2) 

! , c(1) 

! , c(1)  
 /   \  
/     \  
\text{Backtrack} d(1) 
        /   
       /    
\text{Backtrack} 
```

Output: 
\[
\begin{align*}
X & = 1 \\
X & = 2 \\
\end{align*}
\]
Cuts

Example 2:

\[ a(X) :- b(X). \]
\[ a(X) :- f(X). \]
\[ b(X) :- g(X), v(X). \]
\[ b(X) :- X=4, v(X). \]
\[ g(1) \cdot g(2) \cdot g(3) \cdot \]
\[ v(1) \cdot \]
\[ v(X) :- f(X) \]
\[ f(5) \cdot \]

?- a(X).
\[ X = 1 ; \]
\[ X = 5 ; \]
\[ No \]
Cuts

Example 2:

\[
\begin{align*}
  a(X) & : = b(X) . \\
  a(X) & : = f(X) . \\
  b(X) & : = g(X),!,v(X) . \\
  b(X) & : = X=4,v(X) . \\
  g(1) & . \\
  g(2) & . \\
  g(3) & . \\
  v(1) & . \\
  v(X) & : = f(X) \\
  f(5) & .
\end{align*}
\]

?- a(X).

X = 1 ;
X = 5 ;
No
Cuts

Example 2:

\[
\begin{align*}
\text{a}(X) &: \text{b}(X). \\
\text{a}(X) &: \text{f}(X). \\
\text{b}(X) &: \text{g}(X),!,\text{v}(X). \\
\text{b}(X) &: X=4, \text{v}(X). \\
\text{g}(1). \text{g}(2). \text{g}(3). \\
\text{v}(1). \\
\text{v}(X) &: \text{f}(X) \\
\text{f}(5).
\end{align*}
\]

?- \text{a}(X).
X = 1 ;
X = 5 ;
No
Cuts - Green Cuts, Red Cuts

- **Green Cut**
  - a cut the prunes part of a Prolog search tree that cannot possibly reach a solution
    - No point going down a path of search when you know there will be no solution found there.
    - used mainly for efficiency’s sake.

- **Red Cut**
  - a cut that prunes a search tree which contains a some solution, and therefore alters the set of possible solutions reachable.
    - Powerful tool, yet fatal and can be confusing
    - Handle with care
**Cuts**

**Example 2:**

\[
a(X) :- b(X) .
\]

\[
a(X) :- f(X) .
\]

\[
b(X) :- g(X),!,v(X) .
\]

\[
b(X) :- X=4,v(X) .
\]

g(1), g(2), g(3),

\[
v(1) .
\]

\[
v(X) :- f(X)
\]

\[
f(5) .
\]

\[- a(X) .
\]

\[
x = 1 ;
\]

\[
x = 5 ;
\]

No

---

**GREEN CUT**

Graphical representation of the example with backtracking points and cut locations.
Cuts

Example:

\[
\begin{align*}
  a(1) & : - b(1). \\
  a(2) & : - e(2). \\
  b(X) & : - !,c(X). \\
  b(X) & : - d(X). \\
  c(X) & . \\
  d(X) & . \\
  e(X) & .
\end{align*}
\]

Output:

\[
\begin{align*}
  \text{X} & = 1 \\
  \text{X} & = 2 \\
  \text{No}
\end{align*}
\]
Here is an example of merging two lists:

```prolog
merge([X|Xs],[Y|Ys],[X|Zs]) :- X < Y, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :- X = Y, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs])   :- X > Y, merge([X|Xs], Ys, Zs).
merge(X, [], X).
merge([], Y, Y).
```

- Under the usage `merge(↓,↓,↑)`, the first three clauses **mutually exclusive**
- No need to try the others, if one of them succeeds.
- This is made explicit by a green cut.
Cuts – Green Cut Example

Here is an example of merging two lists:

merge([X|Xs], [Y|Ys], [X|Zs]) :- X < Y, !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :- X = Y, !, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X > Y, !, merge([X|Xs], Ys, Zs).
merge(X, [], X).
merge([], Y, Y).

- Inserting these cuts does not change the answers to any merge query.
- This is ASSUMING that we ONLY use it in the merge(↓,↓,↑) query.
Here is an example of membership testing:

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

The two definitions are **not mutually exclusive.**

Under the usage:

- `member(\downarrow, \downarrow)`: there may be multiple ‘Yes’ returned:
  - Eg: `member(1, [1, 2, 1, 1, 3])`
  - **Effect: Performs the membership test**

- `member(\uparrow, \downarrow)`: it returns all possible members of the list:
  - Eg: `member(X, [1, 2, 1, 1, 3])`
  - **Effect: Performs selection**
Cuts – Red Cut Example

What happens if we add a cut?

\[
\begin{align*}
\text{member}(X, [X|Xs]) & : \neg. \\
\text{member}(X, [Y|Ys]) & : \neg \text{member}(X, Ys).
\end{align*}
\]

- **Under the usage:**
  - **member(↓,↓):** there may be multiple ‘Yes’ returned:
    - Eg: \text{member}(1, [1, 2, 1, 1, 3])
    - **Effect:** Does the membership test
  - **member(↑,↓):** it returns all possible members of the list:
    - Eg: \text{member}(X, [1, 2, 1, 1, 3])
    - **Effect:** Performs selection

- **Adding a cut**
  - Will reduce the multiple results of **member(↓,↓).**
  - But it will prune away all other results of **member(↑,↓),** hence returning only at most one answer.
Cuts - Summary

- Be careful when using cuts.

- Don’t cut away something which you want prolog to return.

- Cuts are green if they cut away a portion of the search tree which is guaranteed to fail.

- Cuts are red if they cut away a portion of a successful search tree.
Programming Techniques

- Techniques used in prolog programming may be classified into the following basic forms:
  - Generate-and-test
  - Divide-and-Conquer
  - Search
  - Meta-programming
Generate and Test

To find a solution = Generate all possible solutions + test them one by one.

Example: What is the greatest element in the list?

max(X,L) :- member(X,L), greatest(X,L).

member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
greatest(X,[]).
greatest(X,[Y|Ys]) :- X >= Y, greatest(X,Ys).

?- max(X,[1,3,6,2,3,6,5]).

Note: max(↑,↓) :- member(↑,↓), greatest(↓,↓).
Searching

- The eight queens problem

sol([]).
sol([X/Y | R]) :-
sol(R),
member(Y, [1,2,3,4,5,6,7,8]).
noattack(X/Y,R).
noattack(_, []). 
noattack(X/Y, [X1/Y1 | R]) :-
    Y =/= Y1, Y1-Y =/= X1-X, Y1-Y =/= X-X1,
    noattack(X/Y, R).
Meta-programming

- A meta-program is a program that takes another program as data (e.g. compilers)

- Prolog particularly suitable for rapid prototyping (no much emphasis on efficiency)

- Once ideas are developed, re-implementation in a more efficient language
Applications

- Database query language
- Language for processing Grammars
- Expert Systems
- Meta-interpreters
- ...

...
Database Query Language

- Relational database consists of relations.
- A relation is a set of tuples.
- Following is an example schema.
  - employee(Id, LastName, Firstname, Salary)
- Any relation is described by a Prolog predicate.
  - employee(100, jones, john, 25).
  - employee(105, smith, betty, 20).
  - employee(112, smith, betty, 40).
- Above describes a relation via Prolog facts.
EDB, IDB

- Basic database relations described by Prolog facts.
  - Called **extensional database** (EDB)

- Define new relations (for the purpose of query) in terms of basic relations.
  - These are described by Prolog rules
  - Called **intensional database** (IDB)
Example

- EDB: the “employee” relation
- IDB: the “well_paid_emp” relation

```
well_paid_emp(F, L) :- employee(I, L, F, S), S > 25.
```

Query:

```
?- well_paid_emp(F, L).
F = betty
L = smith
```
Recursion adds power

- Base relations defined as Prolog facts (EDB)
- Operations easily expressed as IDB predicates
  - **Union**, **Diff**, **Intersection**, **Selection**, ...
  - `r_union_s(X1,..,Xn) :- r(X1,..,Xn).`
  - `r_union_s(X1,..,Xn) :- s(X1,..,Xn)`
- Since IDB relations are Prolog predicates, they can be recursive.
- This gives a query language more powerful than SQL.
Geneology database

\[
\begin{align*}
\text{anc}(X, Y) & :\,\,\, \text{parent}(X, Y). \\
\text{anc}(X, Y) & :\,\,\, \text{parent}(X, Z), \text{anc}(Z, Y).
\end{align*}
\]

- EDB: parent, IDB: anc
- ?- \text{anc}(X, \text{charles}).
  - Return all ancestor of “charles”
Grammars

\[
\begin{align*}
\langle S \rangle & \ ::= \langle NP \rangle \ \langle VP \rangle \\
\langle NP \rangle & \ ::= \langle Det \rangle \ \langle N \rangle \\
\langle VP \rangle & \ ::= \langle V \rangle \ \langle NP \rangle \ | \ \langle V \rangle \\
\langle Det \rangle & \ ::= \text{the} \ | \ \text{a} \\
\langle N \rangle & \ ::= \text{man} \ | \ \text{woman} \\
\langle V \rangle & \ ::= \text{walks} \ | \ \text{likes}
\end{align*}
\]

- Example sentence:  a man walks.
Grammars

Example
s(S0, S) :- np(S0, S1) , vp(S1, S).
np(S0, S) :- det(S0, S1), n(S1, S).
vp(S0, S) :- v(S0, S1) , np(S1, S).
vp(S0, S) :- v(S0, S).
det([the|S], S).
n([man|S], S)).
v([walks|S], S).
det([a|S], S).
n([woman|S], S)).
v([likes|S], S).

?- s([a, man, walks], []).
Yes
Grammars

- Sentence represented as a Prolog list (of terminals).
- Terminal represented as an atom.
- Each non-terminal described by a Prolog predicate.
  - VP described by `vp(S1, S2)`
  - First argument: Input parameter
    - List representing input string
  - Second argument: Output parameter
    - List yet to be parsed after VP has been recognised.

- Topic of Comp. Linguistics
Extensions to logic programming

- Object-oriented LP
- Fuzzy LP
- Concurrent LP
- Constraint LP