11
Advanced Programming in Prolog

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2021/2022

Outline

Previous week:

- The logic programming (and Prolog) paradigm
- Computational model (resolution + unification)
- Basic, idiomatic programming

This week:

- some more advanced programming with Prolog
- other ADTs and libs

Next week:

- meta-programming
- Java/Scala integration with Prolog, by tuProlog

Outline

On building algorithms

Items to discuss

- performance aspects
- tail, non-tail recursions
- immutability and sharing
- generating combinations and searching space of solutions

Progression

- we will start with lists
- then generalize to other ADTs
- meanwhile present libraries
- meanwhile introduce some non-relational construct/predicate

Performance considerations with Prolog

General aspects

- traditionally, Prolog is known for being not "as fast as" C
- professional implementations are actually rather fast
- tuProlog is not fast, for the JVM extra-layer and because it is not a goal of it

Recall our approach to performance

- address the problem only if we have requirements that are not met
- still, we have to know the implications of our programming choices, and choose slower implementations only if they have other good properties, e.g., simplicity, clarity

How can we characterised the performance of a predicate?

- a reasonable approach: in terms of number of resolution steps
- each step requires:
  - search of matching rule
  - computation of mgu
  - update of resolvent under new substitution
  - ⇒ might assume it is constant
  - ⇒ (it actually depends on the number of rules, of variables, and so on)

The case of built-in lists

The list ADT

- write `[H1,H2,...,Hn|T]` instead of `'.'(H1, '.'(H2, ...,'.'(Hn,T)...))`
- write `[H1,H2,...,Hn]` instead of `[H1,H2,...,Hn|[]]`
- `[H|T]`, `[]`, `[E1,E2]`, `[E1,E2|_]` as example special cases

Functionalities over lists

- all expressed as predicates, used to model I/O behaviour
- the same predicate can often be used both to check properties, extract information, generate lists that match criteria, iterate elements

List predicates in Prolog library

- `member(Element, List)`, similar to our `find/element`
- `append(List1, List2, List)`, similar to our `join`
- `reverse(List, ReversedList)`

Tail recursion, in Prolog

Recall definition of tail recursion

- a recursion is "tail" if the recursive call is the last operation executed before returning

Implication of recursive calls

- with recursive calls, nothing is to be done when the base case is reached, hence computation is done before that, namely, it is done while recurring
- hence, optimisation can (at least in principle) be put in place to avoid the cost of creating activation records for the nested calls
- often (not always), tail recursions are to be achieved by putting extra-arguments in the call, modelling state evolution during recursion

Tail-recursive calls in Prolog

- are featured by the rules where the head predicate occurs only as last goal in the body
- the resolution "chain" is such that computation is done in arguments or substitutions, until the base case is reached
- note that non-tail recursions are often more idiomatic
- Prolog supports tail recursive foldright-like List construction!!

Non-tail recursion with `sum/2` (foldleft-like)

```
% sum(List,Sum)
% relate a List of numbers with the sum of its elements
sum([], 0).
sum([H|T], N) :-  sum(T, N2), N is H + N2.
```

```
?- sum([10,20,30], S).   -> S/60
?- sum([], S).           -> S/0
?- sum([10,20,30], 60).  -> Yes
```

sum([10,20,30], S)

sum([20,30], S'), S is 10 + S'

sum([30], S''), S' is 20 + S'', S is 10 + S'

sum([], S'''), S'' is 30 + S''', S' is 20 + S'', S is 10 + S'

S'' is 30 + 0, S' is 20 + S'', S is 10 + S'

S' is 20 + 30, S is 10 + S'

S is 10 + 50

$\{S/60\}$

Notes

- the program is quite idiomatic
- computation occurs as recursion is over
- many resolutions steps are needed ($\sim 2 * n$).

Tail recursion with sum/2 (foldleft-like)

```
% sum(List,Sum)
% relate a List of numbers with the sum of its elements
sum(L, S) :- sum(L, 0, S).
sum([], S, S).
sum([H|T], N, S) :- N2 is H + N, sum(T, N2, S).
```

```
?- sum([10,20,30], S).   -> S/60
?- sum([], S).           -> S/0
?- sum([10,20,30], 60).  -> Yes
```

sum([10,20,30], S)

sum([10,20,30], 0, S)

N2' is 0+10, sum([20, 30], N2', S)

sum([20, 30], 10, S)

N2" is 10+20, sum([30], N2", S)

sum([30], 30, S)

N2"' is 30+30, sum([], N2"", S)

sum([], 60, s)

$\{S/60\}$

Notes

- the program is a bit less direct
- computation occurs "during" recursion
- less resolutions steps are needed ($\sim n * 2$).

Tail recursion with `join/3` (foldright-like)

```
% join(List1, List2, List)
% relate List1 and List2 with their concatenation
join([], L, L).
join([H | T], L, [H | T2]) :- join(T, L, T2).
```

```
?- join([10,20],[30,40,50],L). -> L/[10,20,30,40,50]
```

join([10,20],[30,40,50],L).

join([20],[30,40,50],T2'):$\{L/[10|T2']\}$

join([],[30,40,50],T2''):$\{L/[10|T2'], T2/[20|T2'']\}$

$\{L/[10|T2'], T2'/[20|T2''], T2''/[30,40,40]\}\equiv\{L/[10,20,30,40,50]\}$

Notes

- foldright-like functions are very easily expressed
- the corresponding recursion is tail!!
- the output list is constructed by `[H|T2]` unification during recursion
- this is not achieved by FP with immutable structures

Immutability and sharing: update/4

```
% update(List1, E1, E2, List2)
% relate List1 with a List2 where first occurrence of E1 is updated with
    E2
update([], _, _, []).
update([E1 | T], E1, E2, [E2 | T]).
update([H1 | T1], E1, E2, [H1 | T2]) :- update(T1, E1, E2, T2).
```

```
?- update([10,20,30,40],20,21,L). -> L/[10,21,30,40]
```

update([10,20,30,40],20,21,L)

update([20,30,40],20,21,T2'):$\{L/[10|T2']\}$

$\{L/[10|T2'], T2'/[21|[30,40]]\} \equiv \{L/[10,21,30,40]\}$

Notes

- how is the output list related to the input one?
- Prolog has immutability of data (terms get unified, never modified)
- the output list actually shares `[30,40]` with the input list

⇒ hence Prolog has intrinsic immutability and sharing of structures

Immutability of Prolog terms

Implication of resolution/unification

- computation happens only by resolution + unification
- hence, data values, which are terms, have no concept of mutability
- terms are just manipolated by unification of:
  - terms in the resolvent
  - terms occurring in cloned copies of applied rules

Immutability of terms

- hence, computationally, a term is an entity that never changes
- its subparts could be shared with other terms
- a weak form of side-effect is that by unification a non-ground term could at some point have a variable be "bound" to an actual term

Outline

Full relationality and space-searching

Recall informal definition

- a Prolog predicate is said to be *fully relational* if all its arguments could be handled as either input or output
- most specifically, when called with a variable in an argument, resolution successfully attempts to iterate over all inputs that would satisfy the predicate

$\Rightarrow$ often this property cannot be achieved, especially for complex algorithms

Implication

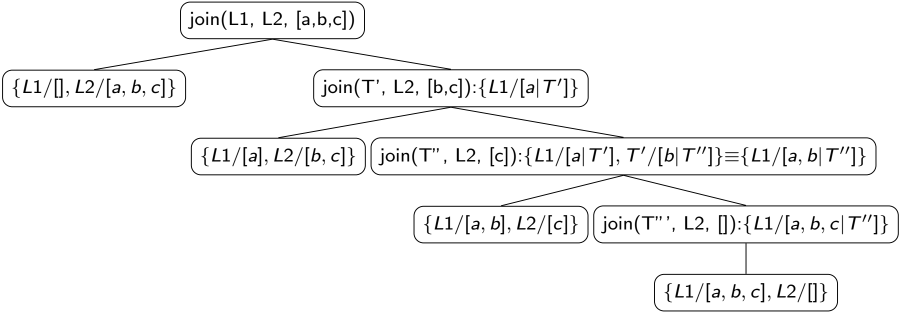
- even if we design a predicate with implicit idea of input arguments and output arguments, a goal could use variables in any place
- so in general we may expect that by resolution Prolog attempts at finding all substitutions of variables satisfying the relation
- this could be automatically obtained in simple cases, or should be explicitly programmed in others

Prolog and solution-space searching

- either way, Prolog is a language with inherent ability of well capturing algorithms that need to search solutions in tree-like spaces

Searching solutions with join/3

```
% join(List1, List2, List)
% relate List1 and List2 with their concatenation
join([], L, L).
join([H | T], L, [H | T2]) :- join(T, L, T2).
```

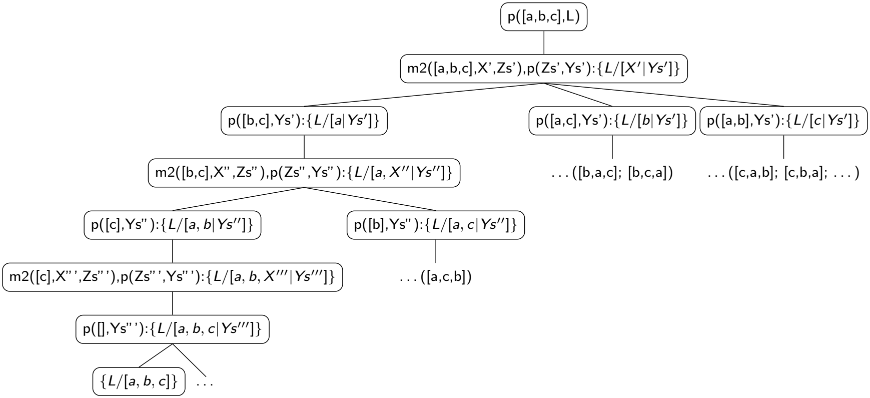


Notes

- we know Prolog will "explore", since the goal matches multiple rules
- thanks to tail recursion, solutions are created "while exploring"

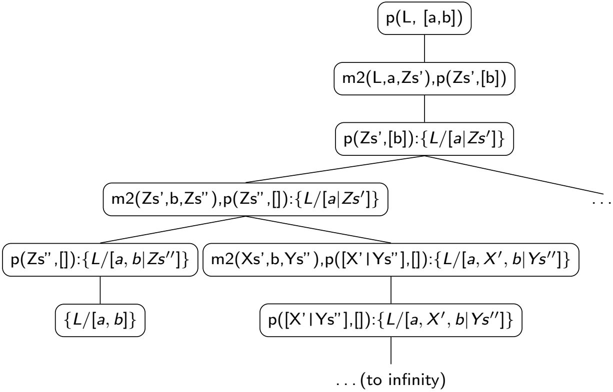
Searching solutions: permutation/2

```
% member2(List, Elem, ListWithoutElem)
member2([X | Xs], X, Xs).
member2([X | Xs], E, [X|Ys]) :- member2(Xs, E, Ys).

% permutation(Ilist, Olist)
permutation([], []).
permutation(Xs, [X | Ys]) :- member2(Xs, X, Zs), permutation(Zs, Ys).
```



Full relationality pitfalls: `permutation(L, [a,b])?`

```
% member2(List, Elem, ListWithoutElem)
member2([X | Xs], X, Xs).
member2([X | Xs], E, [X|Ys]) :- member2(Xs, E, Ys).

% permutation(Ilist, Olist)
permutation([], []).
permutation(Xs, [X | Ys]) :- member2(Xs, X, Zs), permutation(Zs, Ys).
```



Pitfalls in searching

In the general case

- Sometimes building fully relational predicates is not very easy
  - especially with possibly infinite solutions
  - it is not generally easy to enumerate them all
- This is typically the case when enumeration is not strictly directional
- Example:
  - checking if a list has elements 10, 20 and 30...
  - is much easier than enumerating all lists having 10, 20 and 30...

Possible solution in principle

- in certain applications it would still be possible to iteratively extract what needed
- essentially, the idea would be to navigate the resolution tree by breadth-first instead of depth-first strategy
- all solutions would be eventually found, though at very high memory and time costs

I/O notation

Just to be used in documentation: not a Prolog syntax

- Used when describing to a programmer the input/output character of each argument to a predicate
  - "-" for output elements
  - "+" for input elements (of any sort)
  - "@" for input elements that should be ground
  - "?" for input/output elements
- Not very clear how to handle multi-modalities...

Examples

- `member(?E, ?L).`
- `permutation(+LI, -LO).`
- `append(?L1, ?L2, ?L).`

Exploiting exploration to generate combinations

A general pattern in Prolog

- consider a resolvent $\mathtt{G}_1, \mathtt{G}_2, \ldots, \mathtt{G}_n$
- if G_i gives k_i solutions, and they do not constraint execution of successive goals, then overall we would get $\prod_i k_i$ solutions, obtained by all combinations of solutions of each $\mathtt{G}_i$

```
?- member(X,[10,20,30]), member(Y,[1,2]), Res is X+Y.
--> Res/11;
--> Res/12;
--> Res/21;
--> Res/22;
--> Res/31;
--> Res/32;
```

Composition of goals as sort of for-comprehension

- it recalls very much what for-comprehension and `flatMap` is about
- what is also called a *monadic computation*
- so in a sense, Prolog computations are always potentially sorts of for-comprehensions

Generating combinations: links in a grid

```
interval(A, B, A).
interval(A, B, X):- A2 is A+1, A2 < B, interval(A2, B, X).

neighbour(A, B, A, B2):- B2 is B+1.
neighbour(A, B, A, B2):- B2 is B-1.
neighbour(A, B, A2, B):- A2 is A+1.
neighbour(A, B, A2, B):- A2 is A-1.

gridlink(N, M, link(X, Y, X2, Y2)):-
  interval(0, N, X),
  interval(0, M, Y),
  neighbour(X, Y, X2, Y2),
  X2 >= 0, Y2 >= 0, X2 < N, Y2 < M.
```

```
?- gridlink(3,3,L).
--> L/link(0,0,0,1);
--> L/link(0,0,1,0);
...
--> L/link(2,2,2,1);
--> L/link(2,2,1,2)
```

Outline

The cut predicate

Limits of resolution

- the pervasive branching nature of Prolog resolution, along with backtracking, are considered one of the "features" of Prolog
- but in certain situations, they are "a bug"
- certain predicates have spurious solutions one wants to neglect
- handling branching situations in certain predicates violate DRY, and can cause performance issues

Controlling the resolution tree

- Prolog offers extra-relation predicates to get / control "how many" or "which" solutions one wants to extract from a goal
- a very important one is called "cut", performed by 0-ary predicate symbol "`!`"
- its usage is very frequent, and must be well mastered

Cut motivation: dropping spurious solutions

```
% merge(List1,List2,OutList)
% merge two sorted lists
merge(Xs, [], Xs).
merge([], Ys, Ys).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X < Y, merge(Xs, [Y | Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X >= Y, merge([X | Xs], Ys, Zs).
```

```
?- merge([],[],L).
--> L/[]; L/[]              % two equivalent solutions!
?- merge([10,20],[5,35],L).
--> L/[5,10,20,35]; No      % a spurious (costly) "No" reply
```

Notes

- both facts could match
- when one of the three tests succeeds, we do not need to check any of the others!
- in general, checking unnecessary conditions could lead to useless (possibly long) computations
- we may want to express a pruning of the Prolog resolution tree!

The cut predicate details

Syntax

- simply a 0-ary "!" predicate defined at the library level, to be used as one of the goals in the body of a rule

Intended meaning

- it causes certain local pending branches to be discarded:
  - in successive matching clauses
  - in goals at the left of "!" in current body (if they had other pending solutions)

Precise semantics

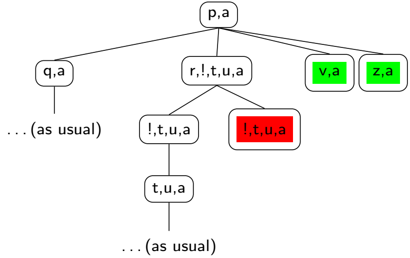
- it is always positively executed, causing a side-effect on the part of the resolution tree yet to be explored
- when a cut is executed, all pending branches below the node that generated the executed cut are "pruned" (i.e., erased away)

What cut prunes

Recall intended meaning

- it causes local pending branches to be discarded:
  - successive matching clauses (green code below)
  - pending solutions in goals at the left of "!" in current body (red code below)

```
r.
r.
p :- q.
p :- r, !, t, u.
p :- v.
p :- z.
```



Cut motivation: solution of merge/3

```
% merge(+List1,+List2,-OutList)
% merge two sorted lists
merge(Xs, [], Xs) :- !.
merge([], Ys Ys).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X<Y, !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- merge([X|Xs], Ys, Zs).
```

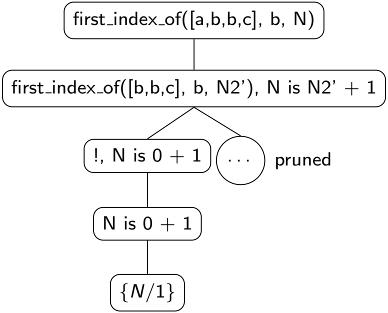
```
?- merge([],[],L).
--> L/[]
?- merge([10,20],[5,35],L).
--> L/[5,10,20,35]
```

Notes

- the first cut prunes the pending branch of second fact
- the second cut prunes the pending branch of second rule
- note that in second rule we do not need to check >= again...

Applications: single result in first_index_of/3

```
first_index_of([E|_], E, 0) :- !.
first_index_of([_|T], E, N) :- first_index_of(T, E, N2), N is N2 + 1.
```

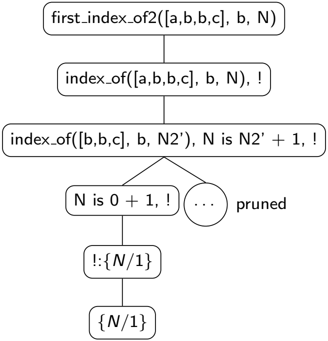


Notes

- as soon as "`!, N is 0 + 1`" moves to "`N is 0 + 1`", all pending branches below "`first_index_of([b,b,c]...`" node get pruned
- this causes activation of second clause to be excluded

Applications: alternative approach

```
index_of([E|_], E, 0).
index_of([_|T], E, N) :- index_of(T, E, N2), N is N2 + 1.
first_index_of2(L, E, N) :- index_of(L, E, N), !.
```



Notes

- as soon as "$!:\{N/1\}$" moves to "$\{N/1\}$", all pending branches below "`index_of([b,b,c], N2', b), ...`" node get pruned
- this causes additional solutions of "`index_of([a,b,b,c], b, N)`" to be excluded

Another example: quicksort

```
% quicksort(Ilist,Olist)
quicksort([], []).
quicksort([X | Xs], Ys):-
  partition(Xs, X, Ls, Bs),
  quicksort(Ls, LOs),
  quicksort(Bs, BOs),
  append(LOs, [X | BOs], Ys).

% partition (Ilist,Pivot,Littles,Bigs)
partition([] , _, [], []).
partition([X | Xs], Y, [X | Ls], Bs):- X<Y, !, partition(Xs, Y, Ls, Bs).
partition([X | Xs], Y, Ls, [X | Bs]):- partition(Xs, Y, Ls, Bs).
```

```
?- partition([10,3,20,5,30,9,40], 10, L1, L2).
--> L1/[3,5,9], L2/[10,20,30,40]
?- quicksort([60,10,20,50,30,40],L).
--> L/[10,20,30,40,50,60]
```

Outline

Library predicates to deal with terms

Managing terms by unification

- so far unification is our only mean to manage terms
- it has been used to inspect terms (to process "input") and to create new terms (to produce "output")
- often, additional expressiveness is needed, though somewhat "extra-relational", as cut

Usual predicates/operators

- inspect shape of a term: `var/1`, `nonvar/1`, `number/1`, `float/1`, `integer/1`, `atom/1`, `compound`, `ground`
- comparison: `=/2`, `\=/2`, `==/2`, `=\=/2`, `copy_term/2`
- terms as strings: `atom_chars/2`, `atom_codes/2`, `atom_concat/2`, `char_code/2`, `number_chars/2`, `number_codes`
- term structure: `=../2`, `functor/2`, `arg/3`
- I/O over file/console: `write/1`, `nl/0`, ...

Some additional documentation:

- https://github.com/tuProlog/2p-kt-presentation/releases/tag/1.2.0-2021-06-28T100324 (2p-kt slides)
- https://github.com/tuProlog/2p-kt/wiki/Prolog-ISO-Standard (prolog ISO)
- https://gitlab.com/pika-lab/tuprolog/2p/-/wikis/Releases (tuprolog-guide-3.3.0)

Inspecting terms

```
?- atom(a).      -> yes
?- atom(10).     -> no
?- atom(a(1)).   -> no

?- var(X).       -> yes
?- var(a).       -> no
?- nonvar(X).    -> no

?- number(10).       -> yes
?- float(10.1).      -> yes
?- integer(10.1).    -> no

?- compound(10).         -> no
?- compound(a).          -> no
?- compound(a(10,20)).   -> yes
?- ground(a(10,20)).     -> yes
?- ground(a(10,X)).      -> no
```

Applications

- typically, to check correctness of inputs

Term comparison

```
% unification
?- a(10,b) = a(10,b).  -> yes
?- a(X,b) = a(10,b).   -> yes X/10
?- a(X,b) = a(Y,c).   -> no

% non-unification (note it never binds)
?- a(10,b) \= a(10,b).  -> no
?- a(X,b) \= a(10,b).   -> no
?- a(X,b) \= a(Y, c).   -> yes

% equality
?- a(10,b) == a(10,b).  -> yes
?- a(X,b) == a(10,b).   -> no
?- a(X,b) == a(X,b).    -> yes

% inequality
?- a(10,b) =\= a(10,b).  -> no
?- a(X,b) =\= a(10,b).   -> yes
?- a(X,b) =\= a(X,b).    -> no

% cloning/check-cloning
?- copy_term(a(10,X), Y).       -> yes, Y/a(10,X1)
?- copy_term(a(10,X), a(10,Y)). -> yes
?- copy_term(a(10,X), a(10,X)). -> yes
?- copy_term(a(10,X), a(11,X)). -> no
```

Applications

- to more flexibly handle/compare inputs

Terms as strings

```
% atoms to/from sequence of one-char atoms
?- atom_chars(hello,L). -> L/[h,e,l,l,o]
?- atom_chars(X,[h,e,l,l,o]). -> X/hello

% atoms to/from sequence of ascii codes
?- atom_codes(hello,L). -> L/[104,101,108,108,111]
?- atom_codes(X,[95,48,32,49]). -> X='_0 1'
?- atom('_0 1'). -> yes

% concatenation
?- atom_concat(aa,bb,X). -> X/aabb

% numbers vs chars/codes
?- number_chars(100,X). -> X/['1','0','0']
?- number_codes(100,X). -> X/[49,48,48]
```

Applications

- to manipulate numbers/atom/strings, computationally

Compound terms (de)structuring

```
% compound term to list
?- p(10,q(20)) =.. L. -> L/[p,10,q(20)]
?- X =.. [p,10,q(20)]. -> X/p(10,q(20))

% direct extraction/construction
?- functor(p(10,20,30),X,Y). -> X/p, Y/3
?- functor(T,p,3). -> T/p(_,_,_)
?- arg(2, p(10,20,30),Y). -> Y/20
```

Applications

- to manipulate compound terms, computationally

I/O over console

Expressiveness

- can redirect standard input/output (is to console by default)
- have predicates to write terms as strings (`write/1`, `nl/0`)
- have predicates to read
- we just see usage for “degugging by logging strings”

```
sum([], 0).
sum([H|T], N) :-  sum(T, N2), write(N2), nl, N is H + N2.
?- write('start'),nl,sum([10,20,30],N).
```

Console

```
start
0
30
50
```

Few examples

```
% all(+Term,+List): are all elements in List of kind Term?
all(_, []).
all(X, [Y | T]):- copy_term(X, Y), all(X, T).

% fully-relational size
size(L, N) :- var(L), !, generate(L, N).
size(L, N) :- length(L, N).

generate([], 0) :- !.
generate([_|T], N) :- N2 is N-1, generate(T,N2).

length([], 0).
length([_|T], N) :- length(T, N2), N is N2 + 1.
```

```
?- all(p(X), [p(a), p(b), p(a)]). --> yes

?- size([10,20,30], N). --> N/3
?- size(L, 3). --> L/[_,_,_]
```

Outline

DB-like structures

The case of DB table operations

- a table of a DB is simply modelled as a list of compound terms
- select, insert, update are managed as expected
- of course performance can be an issue

```
% get_ids(+Table,-List)
% gets the List of ids from the Table
get_ids([], []).
get_ids([user(ID, _, _) | T], [ID | L]) :- get_ids(T, L).

% query(+Table,+Id,-Tuple)
% gets the Tuple with Id from the Table
query([user(ID, N, C) | _], ID, user(ID, N, C)).
query([_ | T], ID, Tuple):- query(T, ID, Tuple).

% update(+Table,+Id,+NewTuple,-NewTable)
% updates the tuple with Id to Tuple
update([user(ID, _, _) | T], ID, Tuple, [Tuple | T]).
update([H | T], ID, Tuple, [H | Table]):-update(T, ID, Tuple, Table).
```

```
?- update([user(100,a,b), user(101,c,d)], 101, user(101,c,e), DB).
-> DB/[user(100,a,b), user(101,c,e)]
```

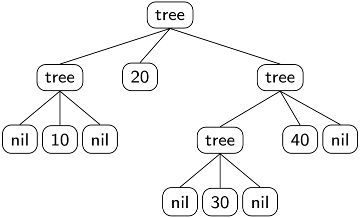
Binary trees: searching elements

Again, a natural modelling

- using functors `tree/3` and `nil/0`
- introducing (relational) operations to find elements

```
% search(+Tree, + Elem), relates a tree with any of its elements
search(tree(_, E, _), E).
search(tree(L, _, _), E) :- search(L, E).
search(tree(_, _, R), E) :- search(R, E).
```

```
?- search( tree(tree(nil, 10, nil), 20, tree(tree(nil, 30, nil),
    40, nil)), E ).
--> E/20; E/10; E/40; E/30
```



Binary trees: other operations

```
% leaves(+Tree,-ListLeaves), returns the list of leaves
leaves(nil, []).        % handling void tree
leaves(tree(nil, E, nil), [E]):- !. % handling a leaf
leaves(tree(L, _, R), O) :- % general case
  leaves(L, OL),      % OL are leaves on left
  leaves(R, OR),      % OR are leaves on right
  append(OL, OR, O).  % O appends the two

% leftlist(+Tree,-List), returns the left-most branch as a list
leftlist(nil, []).
leftlist(tree(nil, E, _), [E]) :- !.
leftlist(tree(T, E, _ ), [E | L]) :- leftlist(T, L).
```

```
?- leaves( tree(tree(nil, 10, nil), 20, tree(tree(nil, 30, nil),
    40, nil)), L).
--> L/[30,40]
?- leftlist( tree(tree(nil, 10, nil), 20, tree(tree(nil, 30, nil),
    40, nil)), L).
--> L/[20,10]
```

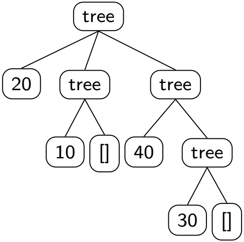
N-ary trees with lists

Again, a natural modelling

- using functors tree/2, with node on first arg, and list of sons in second

```
% searchN(+Tree,?Elem), search Elem in Tree
searchN(tree(E,_), E).
searchN(tree(_,L),E):- member(T, L), search(T, E).
```

```
?- searchN( tree(20, [tree(10, []), tree(40, [tree(30, [])])]), E).
--> E/20; E/10; E/40; E/30
```



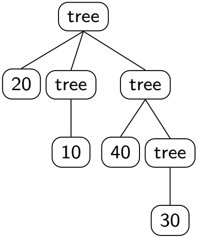
N-ary trees with variable arguments

Modelling

- using functors tree/1, tree/2, tree/3, with node on first arg, and others as sons

```
% searchV(+Tree,?Elem), search Elem in Tree
searchV(T, E) :- T =.. [tree, E | _].
searchV(T, E) :- T =.. [tree, _ | L], member(T2, L), searchV(T2, E).
```

```
?- searchV( tree(20, tree(10), tree(40, tree(30))), E).
--> E/20; E/10; E/40; E/30
```



Bidirectional lists

Design sketch

- constant time to move next/prev on the list
- idea: list `(1,2,3,4,5,6)` modelled by term `bilist([1],[2,3,4,5,6])`
- i.e.: two lists, one from pointer to left, one from pointer to right

Sequence of operations:

- start: `bilist([1],[2,3,4,5,6])`
- move right: `bilist([2,1],[3,4,5,6])`
- move left: `bilist([1],[2,3,4,5,6])`
- addleft(0): `bilist([0],[1,2,3,4,5,6])`
- addright(10): `bilist([0],[10,1,2,3,4,5,6])`

Dynamically expanding lists

Lazy structures in Prolog

- non-ground compound terms can be seen as data structures partially completed
- e.g.: `[1,2,3|_]` is a list starting with 1,2,3, and which we can be completed in several ways
- as a concept, could it be used to model lazy lists?

An example application: an expanding cache for factorials

- `factorial(+N,-Out,?Cache)`
- cache is a partial list of known factorials "up to a point", e.g.: `[1,1,2,6,24|_]`
- each call might expand the cache, which is both input and output

Dynamically expanding lists: code

```
% factorial(+N,-Out,?Cache)
% cache is a partial list of factorials [1,1,2,6,24|_]
factorial(N, Out, Cache) :- factorial(N, Out, Cache, 0).

factorial(N, Res, [Res|_], N) :- !, nonvar(Res).

factorial(N, Out, [H, V | T], I) :-
  var(V), !, I2 is I + 1, V is H * I2,
  factorial(N, Out, [V | T], I2).

factorial(N, Out, [_ , V | T], I) :-
  I2 is I + 1, factorial(N, Out, [ V | T ], I2).
```

```
?- C = [1,1,2,6|_], factorial(5, Res, C).
-> Res/120, C/[1,1,2,6,24,120|_]
```