Programming Language Concepts

Logic Programming Paradigm

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği

Outline

Logic Programming Paradigm

- Based on logic and declarative programming
- 60's and early 70's
- Prolog (**Pro**gramming in **log**ic, 1972) is the most well known representative of the paradigm.
- Prolog is based on Horn clauses and SLD resolution
- Mostly developed in fifth generation computer systems project
- Specially designed for theorem proof and artificial intelligence but allows general purpose computation.
- Some other languages in paradigm: ALF, Frill, Gödel, Mercury, Oz, Ciao, λProlog, datalog, and CLP languages

Constraint Logic Programming

- Clause: disjunction of universally quantified literals,

$$\forall(L_1 \vee L_2 \vee \ldots \vee L_n)$$

- A logic program clause is a clause with exactly one positive literal

$$\forall(A \vee \neg A_1 \vee \neg A_2 \ldots \vee \neg A_n) \equiv$$
$$\forall(A \Leftarrow A_1 \wedge A_2 \ldots \wedge A_n)$$

- A goal clause: no positive literal

$$\forall(\neg A_1 \vee \neg A_2 \ldots \vee \neg A_n)$$

- Proof by refutation, try to unsatisfy the clauses with a goal clause G. Find $\exists(G)$.
- Linear resolution for definite programs with constraints and selected atom.

What does Prolog look like?

```
father(ahmet,ayse).
father(hasan,ahmet).
mother(fatma,ayse).
mother(hatice,fatma).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
```

- CLP on first order terms. (Horn clauses).
- Unification. Bidirectional.
- Backtracking. Proof search based on trial of all matching clauses.

Prolog Terms

- Every valid phrase in prolog is a Term and instead of strict type checking, unification is used. A term can be one of the following:
- Atoms:
  1. Strings with starting with a small letter and followed by any letter, digit and _. `[a-z][a-zA-Z_0-9]*`
  2. Strings consisting of only punctuation symbols as: `[+-*/\^<>=~:.?@#$!&]+`
  3. `[]`, `{}`, `;`, `!` are only treated as atoms in these forms (alone and only spaces in between).
  4. Any string enclosed in single or back quotes. Quotes are not part of the atom.
- Numbers
  1. Any integer `[0-9]+`
  2. Any floating point value `[0-9]+.[0-9]+`
  3. Any scientific notation value `[0-9]+.[0-9]+e[0-9]+`

- Variables:
  1. Strings with starting with a capital letter or _ and consist of `[_A-Z][a-zA-Z_0-9]*`
  2. _ alone is the universal match symbol. Not variable
- Structures:
  - starts with an atom head. No number, no variable
  - has one or more arguments enclosed in paranthesis, separated by comma
  - no space between structure head and paranthesis.
  - arguments can be any valid prolog term, including other structures.

Term	Atom	Num.	Var.	Struct.	not a term
hELLO					
Hello					
_abc					
'A␣and␣B'					
"hello"					
:->>					
:-P					
__					
1e1					
1.0e1					
0.2					
.2					
3.					
0000123					
x(4)					
++(a,b)					
R(3)					
2(4)					
a(a(a,a(a(a(a)))))					
a(X,Y,.(X),2,3)					

Term	Atom	Num.	Var.	Struct.	not a term
hELLO	√				
Hello			√		
_abc			√		
'A␣and␣B'	√				
"hello"				√	
:->>	√				
:-P					√
__			√		
1e1		√			
1.0e1		√			
0.2		√			
.2					√
3.					√
0000123		√			
x(4)				√	
++(a,b)				√	
R(3)					√
2(4)					√
a(a(a,a(a(a(a)))))			√		
a(X,Y,.(X),2,3)				√	

Syntax Elements

- A Prolog program consists of clauses or predicates.
- Unit clauses are structure or atom terms followed by a dot.

```
father(ayse, ahmet).
```

- Unit clauses are considered as facts, no implication.
- Non-unit clauses consists of a head clause and a body

```
grand(X, Y) :- parent(X,Z), parent(Z,X).
```

- In order to prove head clause, body should be proven.
- Body consist of structures seperated by comma, semi-colon and optionally combined with parantheses.

```
uncle(X,Y) :- (brother(X,Z),father(Z,Y)) ; (brother(X,Z), mother(Z,Y)).
```

- Comma stands for conjunction ($\wedge$), semicolon stands for disjunction ($\vee$).
- Structures in the body are goal clauses to be proven.

Prolog Lists

- `[1,2,3]` is parsed and interpreted as `.(1,.(2,.(3,[])))`
- `[Head | Tail]` form is interpreted as `.(Head , Tail)`
- `[]` denotes empty list
- `[1,2,3 | R]` is interpreted as `.(1, .(2, .(3, R)))`
- strings in double quotes like `"abc"` are interpreted as list of ASCII numbers as `[97, 98, 99]`.
- As Prolog structures can contain arbitrary terms, lists are heteregenous as `[1, 2.1, a(b,c), [a,b,c],"hello"]` is a valid list.

Unification

- In functional languages, caller arguments are pattern-matched against the function definition. This operation is also called unification. All constructors and values in caller are matched against the patterns in the definition. The variables in definition are instantiated with the values in the caller.
- Unification in Prolog is bi-directional. Both the defining clause and goal clause have variables instantiated.

  `same(X,X).`

  goal: `same(ali,Y).`

  `X = ali, Y = X, Y = ali`
- Result of a unification can result in some variable instantiations as:

  `X = ali, Y = ali` $\Rightarrow$ `true`

Unification of Terms

unification of x and y is successfull, $x = y \Leftrightarrow$

1. x is atom or number and y is the same atom or number

Unification of Terms

unification of x and y is successfull, $x = y \Leftrightarrow$

1. x is atom or number and y is the same atom or number
2. x, and y are structures with same arity n,
   $x = h_x(x_1, x_2, ..., x_n)$, $y = h_y(y_1, y_2, ..., y_n)$ and
   $h_x = h_y$ and $\forall x_i = y_i,\ i = 1, ...n$. Head and all coressponding elements are unified.

Unification of Terms

unification of x and y is successfull, $x = y \Leftrightarrow$

1. x is atom or number and y is the same atom or number
2. x, and y are structures with same arity n, $x = h_x(x_1, x_2, ..., x_n)$, $y = h_y(y_1, y_2, ..., y_n)$ and $h_x = h_y$ and $\forall x_i = y_i,\ i = 1, ...n$. Head and all coressponding elements are unified.
3. If x is a variable and $x = y$ is compatible with the current set of instantiations, unification is successfull with $x = y$ is added to current set of instantiations.

Unification of Terms

unification of x and y is successfull, $x = y \Leftrightarrow$

1. x is atom or number and y is the same atom or number
2. x, and y are structures with same arity n, $x = h_x(x_1, x_2, ..., x_n)$, $y = h_y(y_1, y_2, ..., y_n)$ and $h_x = h_y$ and $\forall x_i = y_i$, $i = 1, ...n$. Head and all coressponding elements are unified.
3. If x is a variable and $x = y$ is compatible with the current set of instantiations, unification is successfull with $x = y$ is added to current set of instantiations.
4. Otherwise, unification fails.

`a = b`	false
`'abc' = abc`	true
`X = 12`	true $\Leftarrow$ X=12
`a(1,X) = a(Y,2)`	true $\Leftarrow$ X=2, Y=1
`a(1,X) = a(Y,Y)`	true $\Leftarrow$ X=Y=1
`a(1,X,X) = a(Y,Y,2)`	false (Y=1, X=Y, X=2)
`a(1,_) = a(1)`	false (different arities)
`X = a(X)`	true $\Leftarrow$ X = a(X) (cannot display but succesfull)
`a(c(X,d),c(a,Y),X) = a(Y,Z,t)`	true $\Leftarrow$ X = t, Y = c(t,d) , Z = c(a,c(t,d))
`a(c(X,d),c(a,Y)) = a(Y,X)`	true $\Leftarrow$ X = c(a,Y), Y = c(X,d)

Prolog Program

- A Prolog program can be written by putting all alternatives as a seperate head clause with same name and arity.
- You define relations instead of functions returning a value.
- For example, not a membership test function but a member relation.
- Membership relation for a list can be defined verbally as: "*x* is member of list *lst* if either *x* is the first element of the *lst* or it is the member of the remaining list"
- Each alternative is another `member(X,LST)` clause.

```
member(X, [First | Remaining]) :- X = First.
member(X, [First | Remaining]) :- member(X, Remaining).
```

- Shorter form:

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

Prolog Interpreter

- Gnu Prolog or Sicstus Prolog are free alternatives.
- entering '`[filename].`' in interpreter loads the clauses from `filename.pl`.
- '`?- `' prompt asks user to enter goal clauses like:

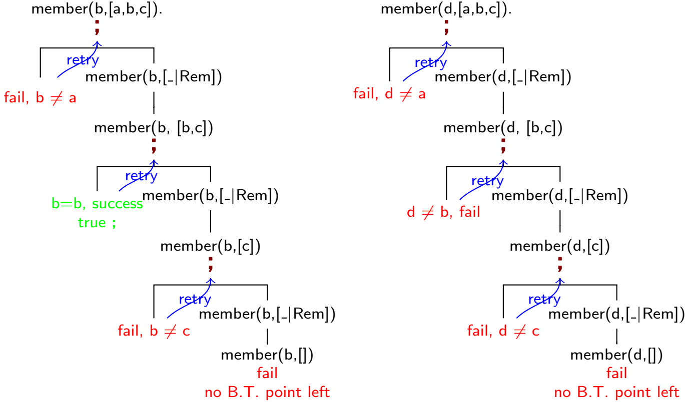
  `?- member(b, [a,b,c]).`
- Prolog checks if this goal can be proven with the current program and replies yes, no
- If there are alternatives, it prompts true ? and asks for continuation. pressing enter will terminate, ; will try other alternatives.

```
~$ swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
...
Please visit http://www.swi-prolog.org for details.

?- [testmember].              % load testmember.pl
true.
?- member(b,[a,b,c]).
true                          % hit enter
?- member(d, [a,b,c]).
false.
?- member(b,[a,b,c]).
true ;                        % hit ; , try alternatives
false.                        % no other alternatives true
?- member(b,[a,b,b]).
true ;                        % hit ; , try alternatives
true ;                        % one more alternative, no other
false.
?- member(X,[a,b,c]).         % ask who is member of [a,b,c]?
X = a ;
X = b ;
X = c ;
false.
```

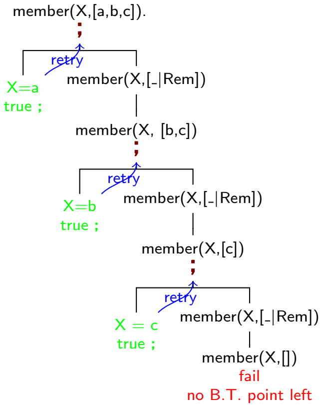
Backtracking

- Backtracking is the search procedure of Prolog and makes it a universal programming language.
- Each alternative head clause that can be unified with goal clause is a backtracking point.
- similarly each operand of ';' is a backtracking point.
- Prolog saves the current state in backtracking points. On failure, tries the next backtraking branch.
- On success, if user hits ';' in prompt, it resumes search from the next backtracking point.

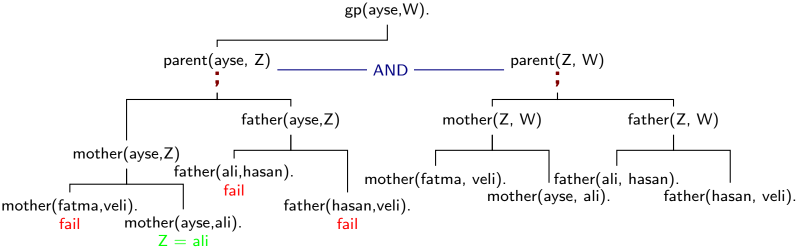
```
member(X, [X | _ ]).

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



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

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

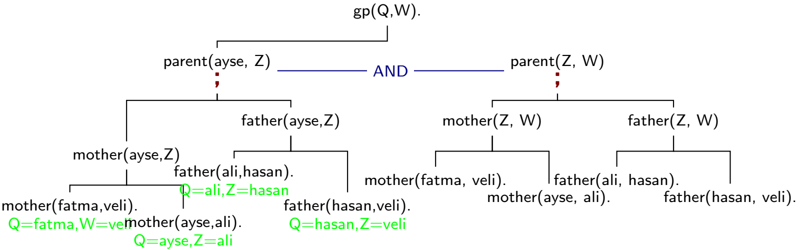


```
mother(fatma,veli).     father(ali, hasan).     parent(X,Y) :- mother(X,Y).
mother(ayse, ali).      father(hasan, veli).    parent(X,Y) :- father(X,Y).
gp(X,Y) :- parent(X,Z), parent(Z,Y).
```



Only solution from left branch is Z=ali, applied to right branch. father(ali,hasan) matches. Result is W = hasan.

```
mother(fatma,veli).     father(ali, hasan).     parent(X,Y) :- mother(X,Y).
mother(ayse, ali).      father(hasan, veli).    parent(X,Y) :- father(X,Y).
gp(X,Y) :- parent(X,Z), parent(Z,Y).
```



For each solution in left parent branch it backtracks and test solution from right parent branch, keeping the instantiated variables. Z=ali and Z=hasan returns success. Results are: Q=ayse,W=hasan and Q=ali,W=veli

List Processing

- Appending lists.

```
append([], LST, LST).
append([H | Rem], LST,) :- append (Rem, LST, Res).
```

- append(X,Y,[a,b,c,d]) works as well.
- Reverse:

```
reverse([], []).      % reverse of empty list
reverse([H|Rem],Rev) :- reverse(Rem, RR), .
```

- Efficient reverse:

```
reverse2([], L, L).      % no element left, result is stack
reverse2([H | Rem],P, L) :- reverse2(Rem, [H | P], L). % inser
reverse(LST, LSTREV) :- reverse2(LST, [], LSTREV).
```

List Processing

- Appending lists.

```
append([], LST, LST).
append([H | Rem], LST,[H | Res]) :- append (Rem, LST, Res).
```

- `append(X,Y,[a,b,c,d])` works as well.
- Reverse:

```
reverse([], []).      % reverse of empty list
reverse([H|Rem],Rev) :- reverse(Rem, RR), .
```

- Efficient reverse:

```
reverse2([], L, L).      % no element left, result is stack
reverse2([H | Rem],P, L) :- reverse2(Rem, [H | P], L). % inser
reverse(LST, LSTREV) :- reverse2(LST, [], LSTREV).
```

List Processing

- Appending lists.

```
append([], LST, LST).
append([H | Rem], LST,[H | Res]) :- append (Rem, LST, Res).
```

- `append(X,Y,[a,b,c,d])` works as well.
- Reverse:

```
reverse([], []).      % reverse of empty list
reverse([H|Rem],Rev) :- reverse(Rem, RR), append(RR,[H], Rev).
```

- Efficient reverse:

```
reverse2([], L, L).     % no element left, result is stack
reverse2([H | Rem],P, L) :- reverse2(Rem, [H | P], L). % inser
reverse(LST, LSTREV) :- reverse2(LST, [], LSTREV).
```

Arithmetical Operations

- `X = 3 * 5` is equivalent to `X = *(3,5)` and does not make any calculation.
- A special operator '`is`' evaluates the expressions:
  `X is 3 * 5` will instantiate X to 15.
- `is` requires right handside to be fully instantiated (no variables without a value) and evaluates it, the resulting number is unified with LHS.
- '`2+X is 5`' is equivalent to unification of `+(2,X)` to 5, which fails.
- Comparison operators also evaluate their both operands which should be fully instantiated:
  `<` , `>` , `=<` , `>=`, `=:=` , `=\=`
- Some of the arithmetic operators (in evaluation context):
  `+`, `+`, `*`, `/` , `//` (int.div.) `mod` .
- Also mathematical functions can be used:
  sin , cos, . . . , exp, log, log10, abs, round, ceil,. . .

Build-in Predicates

- Testing term type:

  `var(T), nonvar(T), atom(T), number(T), atomic(T), ground(T).`

- Equivalance which does not cause instantiation:

  `X == X` strict, `X \== Y` strict not eq., `X \= Y` not unifiable.

- Bidirectional list to term conversion:

  `X =.. [+,b,c]` → `X = +(b,c)`

  `f(a,b,c) =.. X` → `X = [f,a,b,c]`

  `X =.. [t]` → `X = t`

- List predicates:

  `member/2, length/2, append/3, select/3, union/3, reverse/2`

- Displaying all clauses with given name and arity:

  `listing(father/2) listing(reverse/_).`

- Find all solutions in a list:

  `findall(X, father(ali,X), L), setof(X, father(ali,X), L).`

Functional to Logical

- A function can be converted into a relation by adding a result argument. Result can be propagated from recursive calls in this argument.
- Haskell:

```
length [] = 0
length (_:r) = (length r) + 1
```

- Prolog:

```
length([], Res) :- Res is 0.
length([_|R], Res) :- length(R, RLen), Res is RLen + 1.
```

Examples: List

- Relations may work different ways. Give all partitions of a list:

  append(P1, P2, [1,2,3]).

  P1=[],P2=[1,2,3]; P1=[1],P2=[2,3]; P1=[1,2],P2=[3]; P1=[1,2,3],P2=[].

- Selecting/removing element from a list1 results in list2:

```
select(E, [E|R],R).
select(E, [H|R], [H|R2]) :- select(E, R, R2).
```

  select (3,[1,2,3], L)

  L=[1,2]

- For all elements of list1, give remaining list as well:

  select (A,[1,2,3], L)

  A=1,L=[2,3]; A=2,L=[1,3]; A=3,L=[1,2]

- Inserting element to all positions of list1 results in list2:

  select (a,[1,2,3], L)

  L=[a,1,2,3]; L=[1,a,2,3]; L=[1,2,a,3]; L=[1,2,3,a]

- Subset:

```
subset([],[]).
subset(Rsub,[_|R]) :- subset(Rsub,R).
subset([H|Rsub],[H|R]) :- subset(Rsub,R).
```

- Permutations:

```
% insert H to all positions in the remainder permutations
perm([],[]).
% get first element H, get perms of rest
% insert H in every position of rest perm
perm([H|R], HINS) :- perm(R,RP), select(H, HINS, RP).
```

- N combinations:

```
combin(_, [], 0). % 0 combination is empty
% all N combin of remain. is also in comb.
combin([_|R], Res, N) :- N > 0, combin(R,Res,N).
% N-1 combin of remain. add H
combin([H|R], [H|Res], N) :- N > 0, M is N-1,
            combin(R,Res,M).
```

- N permutations:

```
permut(_, [], 0). % 0 permutation is empty
% for all elements H of L, permute remaining.
permut(L, [H|RP], N) :- N > 0, M is N-1,
          select(H, L, Rem), permut(Rem, RP, M).
```

- `\+( P )` or `not(P)` is negation as failure operator. Successfull only if the argument clause fails (cannot be proven).
- Set intersection:

```
inter([],_,[]).
inter([H|R],S, [H|Res] ) :- member(H,S), inter(R,S,Res).
inter([H|R],S, Res ) :- not(member(H,S)), inter(R,S,Res).
```

- Set union:

```
union([],S,S).
union([H|R],S, Res ) :- member(H,S), union(R,S,Res).
union([H|R],S, [H|Res] ) :- not(member(H,S)), union(R,S,Res).
```

Cut

- f(x) = if x > 10 then 5 else if x > 5 then 3 else 1

```
f(X, Y) :- X > 10, Y = 5.
f(X, Y) :- X =< 10, X > 5, Y = 3.
f(X, Y) :- X =< 5, Y = 1.
```

Cut

- f(x) = if x > 10 then 5 else if x > 5 then 3 else 1

```
f(X, Y) :- X > 10, Y = 5.
f(X, Y) :- X =< 10, X > 5, Y = 3.
f(X, Y) :- X =< 5, Y = 1.
```

- Each clause test for interval but only one clause can be true.

Cut

- f(x) = if x > 10 then 5 else if x > 5 then 3 else 1

```
f(X, Y) :- X > 10, Y = 5.
f(X, Y) :- X =< 10, X > 5, Y = 3.
f(X, Y) :- X =< 5, Y = 1.
```

- Each clause test for interval but only one clause can be true.
- Cut symbol, '!' prunes search tree and change behaviour.

```
f(X, Y) :- X > 10, ! , Y = 5.
f(X, Y) :- X > 5, ! , Y = 3.
f(X, Y) :- Y = 1.
```

Cut

- f(x) = if x > 10 then 5 else if x > 5 then 3 else 1

```
f(X, Y) :- X > 10, Y = 5.
f(X, Y) :- X =< 10, X > 5, Y = 3.
f(X, Y) :- X =< 5, Y = 1.
```

- Each clause test for interval but only one clause can be true.
- Cut symbol, '!' prunes search tree and change behaviour.

```
f(X, Y) :- X > 10, ! , Y = 5.
f(X, Y) :- X > 5, ! , Y = 3.
f(X, Y) :- Y = 1.
```

- Cut is always successfull with side effect of deleting all backtracking points from head clause so far. Only current solution is kept.

Cut

- f(x) = if x > 10 then 5 else if x > 5 then 3 else 1

```
f(X, Y) :- X > 10, Y = 5.
f(X, Y) :- X =< 10, X > 5, Y = 3.
f(X, Y) :- X =< 5, Y = 1.
```

- Each clause test for interval but only one clause can be true.
- Cut symbol, '!' prunes search tree and change behaviour.

```
f(X, Y) :- X > 10, ! , Y = 5.
f(X, Y) :- X > 5, ! , Y = 3.
f(X, Y) :- Y = 1.
```

- Cut is always successfull with side effect of deleting all backtracking points from head clause so far. Only current solution is kept.
- Rewrite set intersection with a cut:

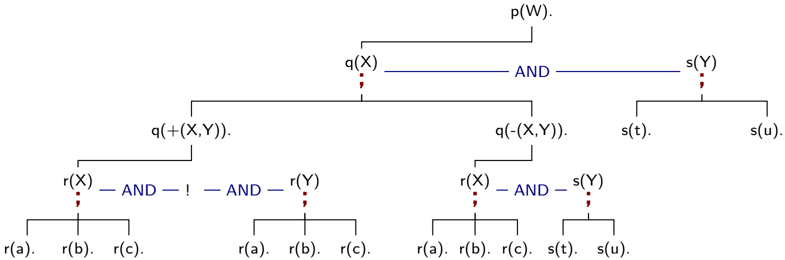
```
inter([],_,[]).
inter([H|R],S, [H|Res] ) :- member(H,S), !, inter(R,S,Res).
inter([H|R],S, Res ) :- inter(R,S,Res).
```

- `\+(P)`, not operator can be implemented by a cut.

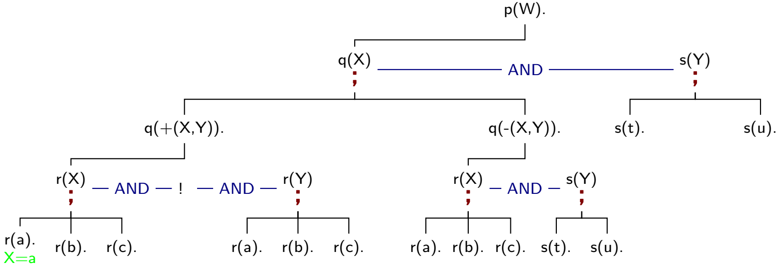
```
not(P) :- P , !, fail.
not(P).
```

- This is called negation as failure semantics, not logical negation. In logical negation you may expect `not(member(X,[a,b,c]))` to instantiate X to complement set of `[a,b,c]`. However it simply fails.
- When a cut does not change the program semantics, set of values returned, it is called a green cut.

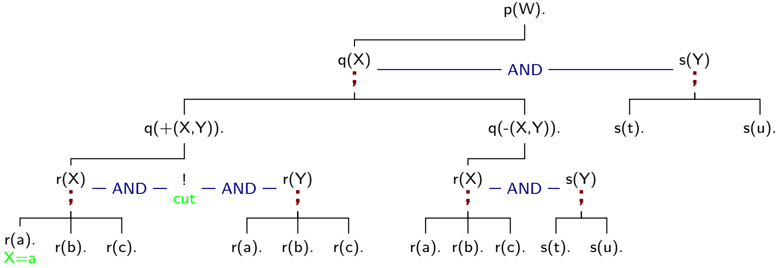
```
p(*(X,Y)) :- q(X), s(Y).      r(a).    s(t).
q(+(X,Y)) :- r(X), !, r(Y).   r(b).    s(u).
q(-(X,Y)) :- r(X), s(Y).      r(c).
```



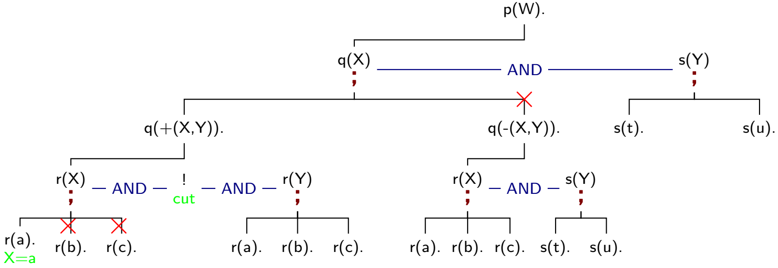
```
p(*(X,Y)) :- q(X), s(Y).      r(a).    s(t).
q(+(X,Y)) :- r(X), !, r(Y).   r(b).    s(u).
q(-(X,Y)) :- r(X), s(Y).      r(c).
```



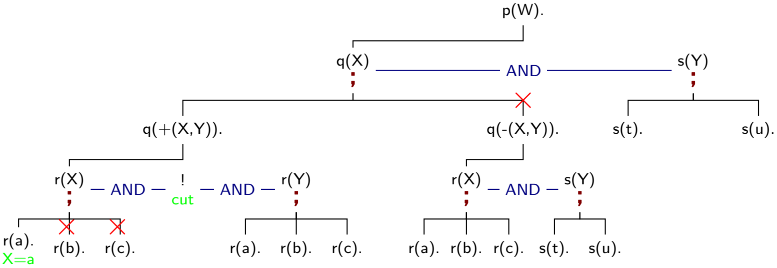
```
p(*(X,Y)) :- q(X), s(Y).      r(a).    s(t).
q(+(X,Y)) :- r(X), !, r(Y).   r(b).    s(u).
q(-(X,Y)) :- r(X), s(Y).      r(c).
```



```
p(*(X,Y)) :- q(X), s(Y).        r(a).    s(t).
q(+(X,Y)) :- r(X), !, r(Y).     r(b).    s(u).
q(-(X,Y)) :- r(X), s(Y).        r(c).
```



```
p(*(X,Y)) :- q(X), s(Y).        r(a).    s(t).
q(+(X,Y)) :- r(X), !, r(Y).     r(b).    s(u).
q(-(X,Y)) :- r(X), s(Y).        r(c).
```



- When cut is hit, current solution is kept and all backtracking points from head clause to cut are pruned. Backtracking points introduced later still prouces alternatives.
- r(Y) produces 3 alternatives, s(Y) at top right produces 2. Prolog finds 6 solutions in total.
- Without a cut: $3 \times 3 = 9$ from q(+(X,Y)), $3 \times 2 = 6$ from q(-(X,Y)) give 15 solutions for q(X). 15 times 2 solutions for s(Y) gives 30 solutions.

The following program generates 90 alternatives. Putting cut in marked positions one at a time changes this behaviour.

```
p( +(X,Y,Z) ) :- ○ q(X) ,○  r(Y),○  s(Z) ○. % 15*3*2 = 90

% 15 for q(X)
q( -(X,Y) ) :-○  r(X),○  r(Y)○  .  %  3 * 3 = 9
q( -(X,Y) ) :-○  r(X),○  s(Y)○  .  %  3 * 2 = 6

r(a).  % 3 from r(X)
r(b).
r(c).

s(t). % 2 from s(X)
s(u).
```

Number of solutions per cut will be:

The following program generates 90 alternatives. Putting cut in marked positions one at a time changes this behaviour.

```
p( +(X,Y,Z) ) :- ○ q(X) ,○  r(Y),○  s(Z) ○. % 15*3*2 = 90

% 15 for q(X)
q( -(X,Y) ) :-○  r(X),○  r(Y)○  .  %  3 * 3 = 9
q( -(X,Y) ) :-○  r(X),○  s(Y)○  .  %  3 * 2 = 6

r(a).  % 3 from r(X)
r(b).
r(c).

s(t). % 2 from s(X)
s(u).
```

Number of solutions per cut will be:

90, 6, 2, 1,

54, 18, 6,

90, 66, 60

Example: Binary Search Tree

- we can represent a tree as a Prolog structure. Each node contains a key value pair in `p(k,v)`.

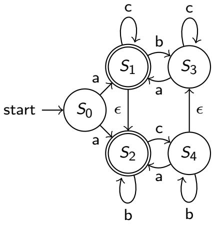
```
insert(e, K,V, tree(p(K,V),e,e)).

insert(tree(p(K,_),L,R), K, V, tree(p(K,V),L,R)) :- !.
insert(tree(p(H,HV),L,R), K, V, tree(p(H,HV),LRes,R)) :-
        K < H ,!, insert(L, K, V, LRes).
insert(tree(p(H,HV),L,R), K, V, tree(p(H,HV),L,RRes)) :-
        insert(R, K, V, RRes).

insertlist(R,[],R).
insertlist(R,[K,V|L], RR) :- insert(R,K,V,RTemp),
                             insertlist(RTemp,L,RR).

search(tree(p(K,V),_,_), K, V) :- ! .
search(tree(p(H,_),L,_), K, V) :- K < H, !, search(L,K,V).
search(tree(p(_,_),_,R), K, V) :- search(R,K,V).

?- insertlist(e,[4,veli,1,ali,6,hasan,2,ayse,5,fatma],R).
R = tree(p(4, veli), tree(p(1, ali), e, tree(p(2, ayse), e, e)
                     tree(p(6, hasan), tree(p(5, fatma), e, e)
```

N-Queens

- Place N queens on a N by N board with no queen is threatening others

```
% fill a list with values from M to N
getMN(N,N,[N]).
getMN(N,M,[N|R]) :- N < M,Nplus1 is N+1,getMN(Nplus1,M,R).

% get N queens, N columns and place them with state []
% queen id is assume to be the row
queen(N,R) :- getMN(1,N,Queens),getMN(1,N,Positions),
        place(Queens,Positions,[],R).

% get first queen, pick a position, check it is compatible wi
% existing state, place rest with new state
place([],[],L,L).
place([Q|QRest],PList,In,RResult) :- select(P,PList,PRest),
    compatible(Q,P,In), place(QRest,PRest,[Q-P|In],RResult).

% compatibility, test not same column and diagonal.
compatible(_,_,[]).
compatible(K,P,[Kk-Pp|R]) :- P =\= Pp, Kdiff is abs(K-Kk),
        Pdiff is abs(P-Pp), Kdiff =\= Pdiff, compatible(K,P,R)
```

Example: NDFA



- Defined by a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$:
  Q set of states, Σ input symbols,
  $\Delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ set of transitions,
  $q_0 \in Q$ start state, $F \subseteq Q$ final states.
- In prolog, we can define all those relations:
  1. define all transitions as trans(s0, a, s1).
  2. define all empty transitions as empty(s1,s2).
  3. define all accepting states as final (s1).
  4. define starting state as start (s0).
- NDFA parser using backtracking power of Prolog is easy:

```
parse(State, []) :- final(State).
parse(State, [H|R]) :- trans(State, New, H), parse(State,R).
parse(State, Inp) :- empty(State, New), parse(State,Inp).
parse(Inp) :- start(S), parse(S, Inp).
```

Example: Numbers game

- Given list of numbers, i.e. [1,3,5,9,30], find arithmetic operations to calculate the given number, i.e. 331 = (9-3+5)*30+1
- findit (numlist, number, expression)
- Pick two numbers from list, pick an operator, compose an expression and put it back on the remaining list and try with new list. When the expression evaluates to the number, it is a success.

```
% if top element evaluates to number, it is the result
findit([A|_],V,A) :- V is A.
% pick two values, pick one operand, check if valid,
% compose an expression, test with new expression
findit(L,V,T) :- select(A,L,AR), select(B,AR,Rest),
        member(O,[+,-,//,*]), check(O,A,B), E =.. [O,A,B],
        findit([E|Rest],V,T).
check(+,_,_).
check(*,_,_).
check(-,A,B) :- A>B.
check(//,A,B) :- 0 is A mod B.
```

Implementation creates duplicate results because of symmetry in '+' and '*'. In symmetric operators use combination, taketwo which will get a, b once, not b,a.

```
taketwo(A, B, [A|R] , Rem ) :- select(B, R, Rem).
taketwo(A, B, [H|R], [H|Rem]) :- taketwo(A, B, R, Rem).

% if top element evaluates to number, it is the result
findit([A|_],V,A) :- V is A.
% asymmetric ops.
findit(L,V,T) :- select(A,L,AR), select(B,AR,Rest),
        member(O,[-,//]), check(O,A,B), E =.. [O,A,B],
        findit([E|Rest],V,T).
% symmetric ops.
findit(L,V,T) :- taketwo(A,B,L,Rest),
        member(O,[+,*]),  E =.. [O,A,B],
        findit([E|Rest],V,T).

check(-,A,B) :- A>B.
check(//,A,B) :- 0 is A mod B.
```

Example: Symbolic Differentiation

- Given an expression containing the variable, find the derivative of the expression with respect to that variable.
- diff (exp, var, diffexp).
- diff (3*x*x+2x+1, x, D) results in D = 6*x+2..

```
diff(A, A, 1) :- !.    %  x -> 1
diff(A, _, 0) :- number(A).  % c -> 0
diff(-A, X, -C) :- diff(A, X, C).  %  -f(x) = - f(x)
diff(A+B, C, D+E) :- diff(A, C, D), diff(B, C, E).
diff(A-B, C, D-E) :- diff(A, C, D), diff(B, C, E).
diff(A*B, C, A*D) :- number(A), diff(B, C, D),!.
diff(A*B, C, A*D+B*E) :- diff(A, C, E), diff(B, C, D).
diff(A/B, C, D) :- diff(A*B^(-1), C, D).
diff(A^B, C, B*A^B1*D) :- number(B), diff(A, C, D), B1 is B -
diff(log(A), B, C*A^ -1) :- diff(A, B, C).

?- diff(1/((x^2+1)*(x-1)),x,R).
R = 1* (-1* ((x^2+1)* (x-1))^ (-1-1)*
        ((x^2+1)* (1-0)+ (x-1)* (2*x^ (2-1)*1+0))).
```

A simplifier is needed.

```
constant(X) :- number(X),!.
constant(A):- A =.. [_,T1,T2] , constant(T1),constant(T2).
constant(A) :- A =.. [_,T1] , constant(T1).
s(X,Y) :- constant(X),!,Y is X.
s(A*1,Y) :- simp(A,Y),!.
s(1*A,Y) :- simp(A,Y),!.
s(-1*A,-Y) :- simp(A,Y),!.
s(A+0,Y) :- simp(A,Y),!.
s(0+A,Y) :- simp(A,Y),!.
s(A-0,Y) :- simp(A,Y),!.
s(0-A,-Y) :- simp(A,Y),!.
s(A^1,AE) :- simp(A,AE),!.
s(1^_,1) :- !.
s(_^0,1) :- !.
s(X,X).

simp(A*B,R) :- simp(A,AE),simp(B,BE),s(AE*BE,R),!.
simp(A+B,R) :- simp(A,AE),simp(B,BE),s(AE+BE,R),!.
simp(A-B,R) :- simp(A,AE),simp(B,BE),s(AE-BE,R), !.
simp(A^B,R) :- simp(A,AE),simp(B,BE),s(AE^BE,R),!.
simp(X,X).
derivative(X,R) :- diff(X,Z), simp(Z,R).

?- derivative(1/((x^2+1)*(x-1)), x, R).
R = - ((x^2+1)* (x-1))^ -2* (x^2+1+ (x-1)* (2*x)).
```