

# Programming Language Concepts

## Logic Programming Paradigm

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



# Outline

- 1 Introduction
- 2 Prolog basics
- 3 Prolog Terms
- 4 Unification
- 5 Backtracking
- 6 List Processing
- 7 Arithmetical Operations
- 8 List Examples
- 9 Cut

# Logic Programming Paradigm

- Based on logic and [declarative programming](#)
- 60's and early 70's
- Prolog (**P**rogramming in **l**ogic, 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,  $\lambda$ Prolog, datalog, and CLP languages

# Constraint Logic Programming

- Clause: disjunction of universally quantified literals,

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

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

$$\begin{aligned} \forall(A \vee \neg A_1 \vee \neg A_2 \dots \vee \neg A_n) &\equiv \\ \forall(A \Leftarrow A_1 \wedge A_2 \dots \wedge A_n) \end{aligned}$$

- A goal clause: no positive literal

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

- 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)				✓	

# 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 heterogeneous 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
- 2  $x$ , and  $y$  are structures with **same arity**  $n$ ,  
 $x = h_x(x_1, x_2, \dots, x_n)$ ,  $y = h_y(y_1, y_2, \dots, y_n)$  and  
 $h_x = h_y$  and  $\forall x_i = y_i$ ,  $i = 1, \dots, n$ . 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.

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

# Prolog Program

- A Prolog program can be written by putting all alternatives as a separate 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.

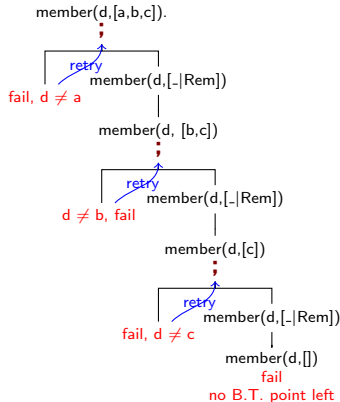
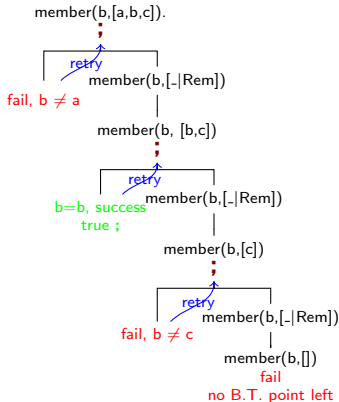
```

# 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 backtracking branch.
- On success, if user hits ';' in prompt, it resumes search from the next backtracking point.

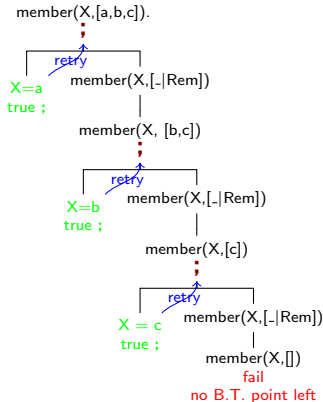
`member(X, [X | _]).`

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



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

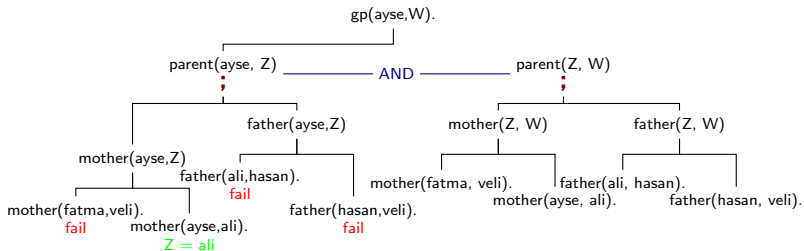
```
member(X, [_ | Rem]) :- member(X, Rem).
```



```

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).

```

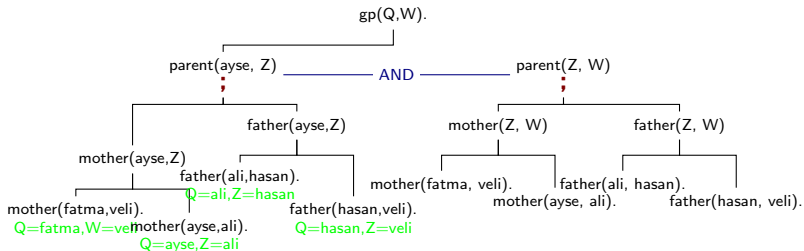


Only solution from left branch is  $Z=ali$ , applied to right branch. `father ( ali , hasan)` matches. Result is  $W = hasan$ .

```

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, [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). % insert
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:  
 $X \text{ is } 3 * 5$  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.
- ' $2+X \text{ is } 5$ ' is equivalent to unification of  $+(2,X)$  to 5, which fails.
- Comparison operators also evaluate their both operands which should be fully instantiated:  
 $< , > , = < , > = , =: = , = \backslash =$
- 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)`.

- Equivalence 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

## ■ Permutations:

```
% insert H to all positions in the remainder permutations
perm([], []).
perm([H|R], HINS) :- perm(R, RP), insertall(H, RP, HINS).

insertall(A, LST, [A|LST]). % insert to beginning
% get first out and insertall positions, put first back
insertall(A, [H|R], [H|RR]) :- insertall(A, R, RR).
```

## ■ Powerset:

```
powerset([], [[]]).
powerset([H|R], Res) :- powerset(R, PR), % w/o H
                        allIns(H, PR, HPR), % with H
                        append(PSR, HPR, Res).

allIns(_, [], []).
allIns(V, [HL | R], [ [V|HL] | Res ]) :- allIns(V, R, Res).
```

## ■ 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
% all N permut of remain. is also in perm.
permut([_|R], Res, N) :- N > 0, permut(R, Res, N).
% N-1 permut of remain. add H in all positions
permut([H|R], Res, N) :- N > 0, M is N-1,
                          permut(R, MP, M), insertAll(H, MP, Res).

```

- `\+( P )` or `not(P)` is **negation as failure** operator. Successful 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) :- \+(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]) :- \+(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.
```

- 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).
```

- $\neg(P)$ , 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**.

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