

Logic Programming and Prolog

Motivation

- Functional Programming involves programming with functions. Substitute a function with its body. Also given a set of input, get an output (left \rightarrow right)
- Logic programming involves programming with relations, where a relation is a set of tuple. n-ary relation: set of n-tuples
- Substitution can work either way in a relation. (left \leftrightarrow right)
- Example: append — a ternary relation on lists:


```

      <math>\langle x, y, z \rangle \in \text{append}</math>
      <math>\langle [], [], [] \rangle \in \text{append}</math>
      <math>\langle [6], [7, 8], [6, 7, 8] \rangle \in \text{append}</math>
      <math>\langle [6, 7], [8], [6, 7, 8] \rangle \in \text{append}</math>
      <math>\langle [7, 8], [6], [6, 7, 8] \rangle \notin \text{append}</math>
      <math>\langle [6, 7, 8], [6], [7, 8] \rangle \notin \text{append}</math>
      
```
- Logic programming involves three elements:
 1. **Terms** that represent entities and relationships among entities
 2. **facts** and **rules** that specify relationships among terms, and
 3. **queries** that ask questions about how terms are related with respect to a collection of facts and rules.

Prolog

Terms

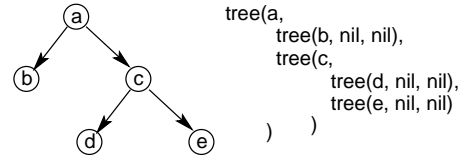
- Facts, rules, and queries are specified using **terms**.
- Simple term:
 1. Atoms: symbolic atoms refer to specific objects. Example: `append`, `parent`, `x`, `y`, `z`
 2. Number: 1, 3, 4, 4.8
 3. Variable: A variable can stand for other values. Variables must begin with an *uppercase*, or the underscore (`_`). Example: `X`, `Variable`, `Y`, `Z`, `_aVar`
- Compound term: Atom followed by a parenthesized sequence of sub-terms. Atom is called **functor** and sub-terms are called arguments.
 - Example:


```
link(bcpl, c)
```

`link`: functor. Note that `link` is not a function.
`bcpl`, `c`: arguments.
 - Compound terms can be nested:

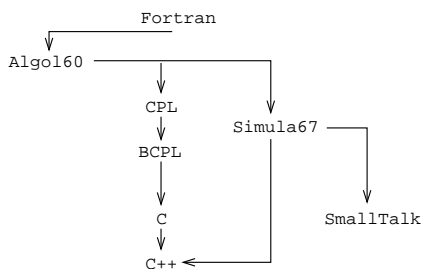

```
tree(a, tree(b, nil, nil),
      tree(c, tree(d, nil, nil),
      tree(e, nil, nil)
      )
      )
```

What this term may mean in real world:



Facts

- **Facts**: A fact expresses *unconditional* relationship among terms.
- Assume the following relationships among various languages:



- Represent the relationship between two languages through term `link`: let `link(x, y)` mean that language `y` is derived from `x`.
- A set of facts that represent the above graph:


```

      link(fortran, algol60).
      link(algol60, cpl).
      link(cpl, bcpl).
      link(bcpl, c).
      link(c, cplusplus).
      link(algol60, simula67).
      link(simula67, cplusplus).
      link(simula67, smalltalk80).
      
```
- `link(bcpl, c)` specifies that `c` is a derivative of `bcpl`.

Rules

- Similar to facts, except that they assert a truth of a relation that is guaranteed only under certain conditions. Logically connects a collection of relations in a conditional form:

How rules are written	What they mean
<code>concl :-</code>	<code>concl is true</code>
<code>cond1,</code>	if <code>cond1 is true</code> and
<code>cond2,</code>	if <code>cond2 is true</code> and
<code>:</code>	<code>:</code>
<code>condN</code>	if <code>condN is true</code>

- Example:


```

      x_grandfather_z :- x_father_y,
      y_father_z.
      x_grandfather_z :- x_father_y,
      y_mother_z.
      
```
- Rule also called a *clause* with a *head* and a *body*
 Head: `x_gradfather_z`
 body: `x_father_y, y_father_z`
- General rules have the following form:


```

      P if (Q1 and Q2 and ... and Qk), where k ≥ 0
      
```

Informal semantics: `P` is true if `Q1` and `Q2`, ..., and `Qk` are true. Formally:

$$P \Leftarrow (Q_1 \wedge Q_2 \wedge \dots \wedge Q_k)$$

Queries

- A functional program is defined by a set of functions. Its execution is driven by evaluation of expressions. A *logic program*, on the other hand, is defined as a set of **rules**; its execution answers a certain **query**.
- Given the database of relationships, query the database regarding the existence of certain relationships.
- A query has the following form: $t_1, t_2, t_3 \dots t_k$.
Formal semantics: $t_1 \wedge t_2 \wedge \dots t_k$
Informally: can we prove that t_1, t_2, \dots , and t_k are true from the database of facts?
- Queries are also called **goals** and individual terms **subgoals**.
- The prolog runtime system will use the database of facts to deduce a goal. If it is successful, it will return **yes**, otherwise **no**.
Yes \Rightarrow Prolog can deduce some fact No \Rightarrow a failure to deduce a fact. Rules cannot be used to deduce negative facts.
- Simple queries:

```
?- link(c, cplusplus).
yes

?- link(algol60, simula67), link(bcpl, c).
yes

?- link(c, bcpl), link(bcpl, c).
no
```

Relations containing variables

- Recall: predicate `x_grandfather_y` used to signify that x is grandfather of y .
Not a very *succinct* way of representing facts; will need to enumerate all such facts one by one. Quite tedious.
- What we need is some way to define relationships among generic terms, that is terms that can stand for other terms.
- **Variables** achieve those.

```
grandfather(X, Z) :- father(X, Y),
                    father(Y, Z).
grandfather(X, Z) :- father(X, Y),
                    mother(Y, Z).
```


Here, X, Y, and Z can take any value.
- Important points:
 - All occurrences of same variable name within a single statement denote the same variable.
 - A fact containing variables asserts that all instances of the fact that result from consistently instantiating all variables are true.
- Example instantiations of variables and their meaning:

```
grandfather(bill, mary) :- father(bill, Y),
                             father(Y, mary).
grandfather(bill, Z)    :- father(bill, Y),
                             mother(Y, Z).
grandfather(john, jim) :- father(john, bob),
                             father(bob, jim).
```
- Anonymous variables(`_`): A variable that takes matches anything, but never takes any value.
Example: check if bill has a father. We don't care who the father is.

```
?- father(bill, _).
```

Unification

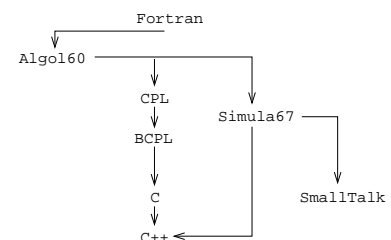
- Logic programming's fundamental operation: How does Prolog bind variables to values?
Unification is an attempt (by Prolog interpreter) to *match* two terms, by instantiating the variables contained in two terms.
- Example: When do two terms $f(X, b)$ and $f(a, Y)$ match?
For $X=a$, and $Y=b$, the two terms are equal.
So unification will involve assigning these values to the variables.
Term $f(a, b)$ is called an *instance* of $f(X, b)$.
- Two terms t_1 and t_2 *unify* if they have a common instance U .
- Where does unification take place?
 1. when two terms are checked if they are equal, through the equality operator
 $f(X, a) = f(b, Y)$
 2. Unification occurs *implicitly* when a rule is applied:

```
identity(Z, Z).
?- identity(f(X, b), f(a, Y)).
X = a
Y = b
```


Note each occurrence of a given variable gets the same instantiation.
 $g(a, a) \in g(X, X)$
 $g(f(a, b), f(a, b)) \in g(X, X)$
 $g(a, b) \notin g(X, X)$

Queries

- Queries with variables:



```
?- link(cpl, X).
X = bcpl;
no
means set of all X such that <cpl, X> ∈ link

?- link(X, cplusplus).
X = C;
X = simula67;
means set of all X such that <X, cplusplus> ∈ link

?- link(X, smalltalk80), link(X, cplusplus).
X = simula67.
means set of all X such that
    <X, smalltalk80> ∈ link ∧ <X, cplusplus> ∈ link

?- link(algol60, X), link(X, Y).
X = cpl
Y = bcpl;

X = simula67
Y = cplusplus;

X = simula67
Y = smalltalk80;
no
```

Queries - cont'd.

- Add more complex rules:

```
path(L, L).
path(L, M) :- link(L, X), path(X, M).
```

1. $\forall L :: \langle L, L \rangle \in \text{path}$ or there is a path from L to L.

2. There a path from L to M if there is a link of L to some node X, and then a path from X to M.

$$(\forall L, M :: \langle L, M \rangle \in \text{path} \Leftarrow (\exists X :: \langle L, X \rangle \in \text{link} \wedge \langle X, M \rangle \in \text{path}))$$

- Queries:

```
?- path(algol60, cplusplus).
yes
```

```
?- path(cplusplus, c).
no
```

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

Data Structures in Prolog

Lists

- Denoted $[a, b, c]$ and $[\]$.

- Head/tail notation (again):

```
[1 | [2, 3]], [1|As], [1, 2 | Bs]
```

- Binding variables to sublists:

```
?- [H|T] = [1, 2, 3].
H = 1      T = [2, 3]
?- [1, 2, 3] = [H|T]
H = 1      T = [2, 3]
?- [1|T] = [H, 2, 3]
H = 1      T = [2, 3]
```

- Real notation for Prolog lists: prefix(.) and $[\]$:

```
[1, 2, 3] sugaring for .(1, .(2, .(3, [])))
```

- Consider term $[1, 2 | X]$. This term denotes the set of all lists that begin with $[1, 2, \dots]$. In Prolog-ese:

$[1, 2 | X]$ is an "open list" and X is its "end-marker" variable. Here X acts as a placeholder.

- Example 1:

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

What if list is nil?

Trace execution of `member(5, [3, 5, 2])`.

- count list elements:

```
length([], 0).
length([_|Tail], K) :- length(Tail, J), K is J+1
```

- Reverse a list:

```
reverse([], []).
reverse([Head|Tail], Result) :- reverse(Tail, ReversedTail),
append(ReversedTail, [Head], Result).
```

Example: Manipulating queues

- Define relation `enter(a, Q, R)` as: when element a enters queue Q, we get queue R.

- Define relation `leave(a, Q, R)` as: when element a leaves queue Q, we get queue R.

- Represent queue by a term $q(L, E)$ where L is an open list with E as an end-marker. Subsequent operations will, in general, extend L.

- Prolog program:

```
setup(q(X,X)).
enter(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
leave(A, q(X,Z), q(Y,Z)) :- X = [A | Y].
wrapup(q([],[])).
```

- How does it work for query:

```
?- setup(Q),
enter(a, Q, R), enter(b, R, S),
leave(X, S, T), leave(Y, T, U),
wrapup(U).
```

- Let's evaluate each:

```
setup(Q) creates an empty queue Q = q(_1, _1)
enter(a, Q, R) creates a new queue R = q([a|_2], _2)
enter(b, R, S) creates a new queue S = q([a, b|_3], _3)
leave(X, S, T) creates a new queue T = q([b | _3], _3)
leave(Y, T, U) creates a new queue U = q(_3, _3)
wrapup(U) => _3 = []
```

```
output:
Q=q([a,b],[a,b]),
R=q([a,b],[b]),
S=q([a,b],[]),
X=a,
T=q([b],[]),
Y=b,
U=q([],[]);
```

Example: Ordered labeled binary trees

- Keys added to left or right according to the value of key.

- Representation:

```
empty atom denoting the empty tree
node(K,S,T) term denoting a tree with label K, and subtrees S and T.
```



- How will a tree be represented?

```
node(2, node(1,empty,empty), node(3,empty,empty))
```

- Creating the tree (insert):

```
insert(K,empty,node(K,empty,empty)).
insert(K,node(N,S,T),node(N,SS,TT)) :-
K<N,insert(K,S,SS).
insert(K,node(N,S,T),node(N,S,TT)) :-
N<K,insert(K,T,TT).
insert(K,node(K,S,T),node(K,S,T)).
?-insert(2,empty,V),insert(3,V,W),insert(1,W,X).
V = node(2,empty,empty)
W = node(2,empty,node(3,empty,empty))
X = node(2,node(1,empty,empty),node(3,empty,empty))
```

- Searching the ordered tree:

```
member(K,node(K,_,_)).
member(K,node(N,S,_)) :- K<N,member(K,S).
member(K,node(N,_,T)) :- N<K,member(K,T).
```

Arithmetic in Prolog

- Prolog defines usual +, -, *, etc operators for performing arithmetic. Use infix notation for specifying expressions.
- An expression is like any other term
 $2 + 3 \equiv +(2, 3)$
- They are not evaluated unless explicitly asked to. Unification rules apply on them as well:

```
?- X = 2 + 3.  
X = 2 + 3
```

Binds X to term 2 + 3.

- In order to evaluate an expression, use `is`:

```
?- X is 2 + 3.  
X = 5
```

- Check the following:

```
?-X is 2+3, X = 2 + 3.  
no
```

term 2+3 does not unify with term 5

- Example 1:

```
sum(1, 1).  
sum(N, Res) :- N > 1,  
               N1 is N - 1,  
               sum(N1, Res1),  
               Res is Res1 + N.
```

```
?- sum(5, X).  
X = 15
```

- Example 2: `close_enough` succeeds if two numbers are equal to within 0.0001.

```
close_enough(X, X) :- .  
close_enough(X, Y) :- X < Y, Y-X < 0.0001.  
close_enough(X, Y) :- X > Y, close_enough(Y, X).
```

Logical operators

- true: goal always succeeds
- fail: always fails
- Equality (=): A term $X = Y$ succeeds if X and Y match. Prolog will try to match them:

```
?- X = 5.  
X = 5.
```

- Inequality(\=): \= is opposite of =
- Negation: The term `not (X)` succeeds if an attempt to satisfy X fails.

```
?- X = 2, not(X = 1).  
X = 2.
```

Unify X with 2 first, and use that X to check if X = 1.

```
?- not (X = 1), X = 2.  
no
```

(X = 1) succeeds by unifying X with 1. Negation term fails causing the whole clause to fail.

- X ; Y: specifies disjunction (or) of goals. The goal succeeds if X succeeds or Y succeeds. Fails when both fail.

```
person(X) :- (X = adam; X=eve; mother(X, Y)).
```

Control in Prolog

- A prolog program contains
 - A sequence of clauses (forming the data base), and
 - a query of the form $A :- B_1, B_2, \dots, B_n$.
- Program evaluation strategy: characterized by two decisions:
 - **Goal order:** Subgoals are processed **left** → **right**
 - **Rule order:** Rules are applied **top** → **bottom**.

Response to a query is affected both by goal order within the query and by rule order within database of facts and rules

- Algorithm for control:

```
Start with a query as the current goal;  
while the current goal is nonempty do  
  let the current goal =  $G_1, \dots, G_k$ , where  $k \geq 1$   
  choose the leftmost goal  $G_1$ ;  
  if a rule applies to  $G_1$  then  
    select first such rule  $A :- B_1, \dots, B_j$ , where  $j \geq 0$   
    let  $\sigma$  be most general unifier of  $A$  and  $B_1$   
    current goal =  $B_1\sigma, B_2\sigma, \dots, B_j\sigma, G_2\sigma, \dots, G_k\sigma$   
  else backtrack;  
end while  
success
```

Example

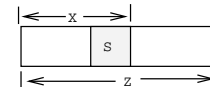
- Example: Let rules be

```
append([], Y, Y).  
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

```
prefix(X, Z) :- append(X, Y, Z).
```

```
suffix(Y, Z) :- append(X, Y, Z).
```

```
sublist1(S, L) :- prefix(X, L), suffix(S, X).  
sublist2(S, L) :- suffix(S, X), prefix(X, L).
```



How does search take place?

- Example query:

```
?- suffix([a], L), prefix(L, [a,b,c]).
L = [a].
```

- Search tree with no backtracking:

```
suffix([a], L), prefix(L, [a, b, c])
↓
suffix([a], L) if append(_1, [a], L).
↓
append(_1, [a], L), prefix(L, [a, b, c])
↓
{ _1->[], L->[a] } append([], [a], [a]).
↓
prefix([a], [a,b,c])
↓
prefix([a],[a,b,c]) if append([a],_2, [a
↓
append([a],_2, [a,b,c])
↓
append([a],_2,[a,b,c]) if append([],_2,[
↓
append([], _2, [b, c])
↓
{ _2->[b,c] } append([], [b,c], [b,c]).
yes
```

How does search take place?

- Example query:

```
?- suffix([b], L), prefix(L, [a,b,c]).
L = [a, b].
```

- Search tree with backtracking:

```
suffix([b], L), prefix(L, [a, b, c])
↓
append(_1, [b], L), prefix(L, [a, b, c])
↓
{ _1->[], L->[b] }
↓
prefix([b], [a,b,c])
↓
append([b],_2, [a,b,c])
↓
backtrack
↓
append(_4, [b], _5), prefix([_3|_5], [a, b, c])
↓
prefix([_3,b], [a,b,c])
↓
append([_3,b],_6, [a,b,c])
↓
{ _3->a }
↓
append([b],_6, [a,b,c])
↓
append([],_6, [c])
↓
{ _6->[c] }
Yes
```

Does goal order matter?

- Order of subgoals within a query affects prolog search tree

```
?- suffix([a], L), prefix(L, [a, b, c]).
L = a ;
[ infinite computation ]
```

- Why? The leftmost subgoal

```
?- suffix([a], L).
L = a ;
L = [_1, a] ;
L = [_1, _2, a] ;
...
```

has an infinite number of solutions, only the first of which satisfies `prefix(L, [a, b, c])`.

The system first finds a solution without backtracking. A futile search through the rest of infinite tree ensues if we ask for a further solution.

- what about reordered query?

```
?- prefix(L, [a, b, c]), suffix([a], L).
L = a ;
no
```

This query has a finite prolog search tree.

Does rule order matter?

- Yes they do since rule orders change the order in which solutions are searched.

```
app2([H|X], Y, [H|Z]) :- app2(X, Y, Z).
app2([], Y, Y).
```

```
?- appen2(X, [c], Z).
```

- Search tree:

```
append(X, [c], Z)
↓
{x->[], z->[c]}
yes
x=[], z=[c]
↓
append(_2, [c], _3)
↓
{ _2->[], _3->[c] }
yes
x=[_1], z=[_1,c]
↓
...
```

```
appen2(X, [c], Z)
↓
{x->[_1|_2], z->[_1|_3]}
↓
appen2(_2, [c], _3)
↓
{x->[], z->[c]}
yes
x=[], z=[c]
↓
{ _2->[], _3->[c] }
yes
x=[_1], z=[_1,c]
```

Cuts

- Cut allows one to prune or “cut out” an unexplored part of a Prolog Search tree.
- Used to specify the following:
 - “if you get this far, you have picked the correct rule for this goal.” Stop looking for anything else.
 - “if you get to here, you should stop trying to satisfy this goal”
 - “if you get here, you have found the only solution of the problems.” Stop looking for any more solutions.

Can therefore be used to make computation more efficient by eliminating futile searching and backtracking.

- A cut written as ! :

$B :- A_1, A_2, \dots, A_m, !, B_1, B_2, \dots, B_n.$

Semantics: Tells control to backtrack past A_1, A_2, \dots, A_m without considering any rules for them.

- Example:

```
member(K,node(K,-,-)).
member(K,node(N,S,-)) :- K<N,!, member(K,S).
member(K,node(N,-,T)) :- N<K, member(K,T).
```

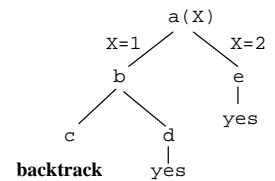
Use cut to specify that if $K \leq N$, and $\text{member}(K, S)$ fails, don't bother to look at the next rule. Without the cut, prolog will backtrack and try the third rule, only to fail on $N < K$.

Example

- Database and search tree for a rule:

```
b :- c.
b :- d.
d.
e.
a(1) :- b.
a(2) :- e.

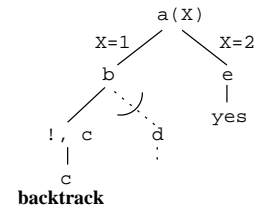
?- a(X).
X = 1;
X = 2;
no
```



- Search tree with cut applied:

```
b :-!, c.
b :- d.
d.
e.
a(1) :- b.
a(2) :- e.

?- a(X).
X = 1;
X = 2;
no
```



Summary

- Computation through manipulation of relationships.
- A prolog program consists of
 - Facts: unconditional relationships among terms
 - Rules: relationships among terms that may be true under certain conditions.
 - Queries that must be true give a set of facts and rules.
- A prolog program merely specifies the various relationships among terms.
- Prolog system defines “how the computation is carried out”
 - Unification of variables and terms: assignment of values to variables so that relationships match.
 - Implement control by searching the different rules: Employ backtracking to search for all solutions.