CS571: Programming Languages

Prolog: PROgramming in LOGic

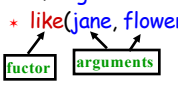
- Developed by Robinson, Colmerauer and Kowalski, in early 70s
- Has been used in
  - Artificial intelligence, e.g. natural language processing, automated reasoning systems, expert systems, etc
  - Security Policy
  - Model Checking
  - ...
- We will use XSB tabled logic programming system
  - Solaris/Linux version: installed on bingsuns
  - http://xsb.sourceforge.net/manual1/manual1.pdf

Terms

- Atoms
  - Integers: 1
  - Floats: 2.31
  - Symbols (begin with lowercase letter) : dog
- Variables (begin with upper case): X, Y
  - `-' can be used in place of a variable if you do not care its value.

Terms (Cont.)

- Structures and Predicates
  - Consists of an atom called the functor and a list of arguments.
  - like(jane, flower)



  - Arguments can be arbitrary terms: constants, variables, or (nested) structures.
  - Arity: number of arguments. The predicate like has arity 2.

Matching Two Terms

- Two Prolog terms t_1 and t_2 unify (i.e. $t_1 = t_2$) if there is some substitution σ (their unifier) that makes them identical: $\sigma(t_1) = \sigma(t_2)$
  - A variable can match another variable
    X = Y with {X → Y}
  - Constants match if they are equal
    susan = susan
  - A variable can match a constant
    X = susan with {X → susan}
  - Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively.
    f(a,g(b,X)) and f(a,g(b,h(c,d)))

Unification: Example

- Are the following true?
  - f(a) = f(X, a)
  - X = f(X)
  - f(X, g(X)) = f(a,Y)

Unification: Example

- Are the following true?
  - f(a) = f(X, a) No
  - X = f(X)
  - f(X, g(X)) = f(a,Y)

Unification: Example

- **Are the following true?**
  - f(a) = f(X, a) No
  - X = f(X) No

    Problem: X = f(f(f(…)))
    Most languages do not have "occur check".
  - f(X, g(X)) = f(a,Y)

Unification: Example

- Are the following true?
  - f(a) = f(X, a) No
  - X = f(X)

    Problem: X = f(f(f(…)))
    Most languages do not have "occur check".
  - f(X, g(X)) = f(a,Y) Yes

    {X → a, Y → g(a)}

Unification: Example

```
?- 2*3 = 6.

?- 2*3 = 2*3.
```


Unification: Example

```
?- 2*3 = 6.
No
?- 2*3 = 2*3.
```


Unification: Example

```
?- 2*3 = 6.
No
?- 2*3 = 2*3.
Yes
```

Boolean Predicates

- A = B unifiable

X = Y ?

- A \= B not unifiable
- A == B identical, does not unify A and B

X == Y ?

- A \== B not identical

Boolean Predicates

- A = B unifiable

X = Y ?

yes

- A \= B not unifiable
- A == B identical, does not unify A and B

X == Y ?

no

- A \== B not identical

Boolean Predicates (Cont.)

- A < B less than (numeric)

1 < 2 yes
abc < bc no
X < Y
Error: Unbound variable in arithmetic expression

- A =< B less or equal (numeric)
- A > B greater than (numeric)
- A >= B greater or equal (numeric)

Boolean Predicates (Cont.)

- A @< B less than (terms)

1 @< 2 yes
abc @< bc yes
X @< Y yes

- A @=< B less or equal (terms)
- A @> B greater than (terms)
- A @>= B greater or equal (terms)

Facts

- The clauses in Prolog database can be classified as facts or rules, each of which ends with a period.
- A prolog program parent.P with 5 facts.

```
father(mike, susan).
father(mike, john).
mother(mary, susan).
mother(mary, john).
mother(jane, mary).
```

- We would naturally interpret these as facts about families: mike is susan's father, …

Facts

- A prolog program parent.P with 5 facts.

```
father(mike, susan).
father(mike, john).
mother(mary, susan).
mother(mary, john).
mother(jane, mary).
```

Load parent.P in XSB Prolog:

```
| ?- [parent].
[Compiling ./parents]
[parents compiled, cpu time used: 0.0810 seconds]
[parents loaded]

yes
| ?-
```

Queries

father(mike, susan).
father(mike, john).
mother(mary, susan).
mother(mary, john).
mother(jane, mary).

- A collection of facts is just a static database.
- Prolog computation is triggered by making queries.
  - Ask whether a relation holds between objects.

| ?- father(mike, susan).
yes
| ?- father(mike, tom).
no

Variables in Queries

- Any term can appear as a query, including a term with variables
- Can be used to look up information.
  - Does there exist an X such that father(X, susan)?

| ?- father(X, susan).
X = mike.
no

ask for more answers
no more answer

  - If more than one X matches, Prolog will return all answers

| ?- father(mike, X).
X = susan.
X = John.
no

  - If you just need one answer, simply press "enter".

| ?- father(mike, X).
X = susan
yes

Variables in Facts, Conjunctive Queries

- Queries can be ANDed together. The Prolog system tries to prove them all using a single set of bindings
  - Who are mike and mary's children?

| ?- father(mike, X), mother(mary, X).
X = susan.
X = john.
no

Rules

Implication symbol

- Rules: deduce new information from old

head → parent(X,Y) :- father(X,Y). condition
Parent(X,Y) :- mother(X,Y).

If father(X,Y) OR mother(X,Y), then parent(X,Y)

To prove the head, prove the condition.

Rules

- Rules: deduce new information from old

parent(X,Y) :- father(X,Y).
Parent(X,Y) :- mother(X,Y).

| ?- parent(X, susan).
X = mike.
X = mary.
no

Universally quantified
Existentially quantified

- Rules can build on rules

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

If parent(X,Z) and parent(Z,Y) then grandparent(X,Y)

Some Features in Prolog

- Assign-once variables: Any variable in a Prolog procedure can only ever get one value assigned to it.
  - has a value which can never be changed
  - Has not yet been assigned a value

| ?- N is 1.
N=1.
No

assignment

| ?- N is 1, N is 2.
No

- All variables are local to the rule in which they occur
- Variables need not be declared and have no type

Some Features in Prolog (Cont.)

- Variables in the predicate may be input or return variables.
- Nondeterminism: you can give multiple definitions of the same procedure

```
parent(X,Y) :- father(X,Y).
Parent(X,Y) :- mother(X,Y).
```


How Prolog Works

How Prolog Works

- Finds solutions by search – goal directed
- Subgoals are considered in left-to-right order
- Clauses are considered in the order of definition. Evaluation depends on ordering of clauses.
- Depth-first search with backtracking
  - If a subgoal fails, Prolog backtracks and tries to resatisfy the subgoal to the left of the one that failed.

```
father(mike, jim).
father(mike, susan).
mother(mary, susan).
| ?- father(mike, X), mother(mary, X).
```



```
father(mike, jim).
father(mike, susan).
mother(mary, susan).
| ?- trace.
yes
[trace]
| ?- father(mike, X), mother(mary, X).
    Call: father(mike,_h90) ?
    Exit: father(mike,jim) ?
    Call: mother(mary,jim) ?
    Fail: mother(mary,jim) ?
    Redo: father(mike,jim) ?
    Exit: father(mike,susan) ?
    Call: mother(mary,susan) ?
    Exit: mother(mary,susan) ?
```


How Prolog Works

```
parents(X,Y) :- father(X,Y).
Parents(X,Y) :- mother(X,Y).
father(tom, jane).
mother(mary, tom).
```

- To prove parents(mary, Y)

1. Select parents(X,Y) :- father(X,Y), and prove father(mary,Y). This proof fails.
2. Go back to step 1, and select the second clause of parents, i.e. parents(X,Y) :- mother(X,Y), and prove mother(mary,Y).
3. Select the fact mother(mary,tom)
4. There is nothing left to prove, so the proof succeeds

Negation As Failure

- To prove not(X), Prolog attempts to prove X
- not(X) succeeds if X fails

```
sibling(X,Y) :-
    not(X=Y),
    parent(P,X),
    parent(P,Y).

parent(mike, kent).

parent(mike, kim).

Parent(susan, mary).

Parent(susan, jean).
```

```
| ?- sibling(kim,kent).

|?- sibling(kim,kim).

?- sibling(X,Y).
```

Negation As Failure

- To prove not(X), Prolog attempts to prove X
- not(X) succeeds if X fails

```
sibling(X,Y) :-
    not(X=Y),
    parent(P,X),
    parent(P,Y).

parent(mike, kent).

parent(mike, kim).

Parent(susan, mary).

Parent(susan, jean).
```

```
| ?- sibling(kim,kent).

yes

|?- sibling(kim,kim).

?- sibling(X,Y).
```


Negation As Failure

- To prove not(X), Prolog attempts to prove X
- not(X) succeeds if X fails

```
sibling(X,Y) :-
    not(X=Y),
    parent(P,X),
    parent(P,Y).

parent(mike, kent).

parent(mike, kim).

Parent(susan, mary).

Parent(susan, jean).
```

```
| ?- sibling(kim,kent).

yes

|?- sibling(kim,kim).

No

?- sibling(X,Y).
```


Negation As Failure

- To prove not(X), Prolog attempts to prove X
- not(X) succeeds if X fails

```
sibling(X,Y) :-
    not(X=Y),
    parent(P,X),
    parent(P,Y).

parent(mike, kent).

parent(mike, kim).

Parent(susan, mary).

Parent(susan, jean).
```

```
| ?- sibling(kim,kent).
yes

|?- sibling(kim,kim).
No

?- sibling(X,Y).

No
```


Negation as Failure (Cont.)

```
sibling(X,Y) :-
    parent(P,X),
    parent(P,Y),

    not(X=Y).

parent(mike, kent).

parent(mike, kim).

Parent(susan, mary).

Parent(susan, jean).
```

```
?- sibling(X,Y).

X = kent
Y = kim.

X = kim
Y = kent .

X = mary
Y = jean .

X = jean
Y = mary .

No
```