CSC326 Programming Languages (Week 11. Friday)

Yilan Gu
guyilan@ecf.toronto.edu
http://www.cs.toronto.edu/~yilan/326f09

Prolog: facts

- **A fact is a clause with an empty body**
- **Syntax**
  <head>.
- **What makes a fact a fact?**
- **Examples**
  - Exams — exams.
  - Assignments — assignments.
  - Taxes — taxes.
  - The earth is round. — round(earth).
  - The sky is blue. — blue(sky).
  - The sun is hot. — hot(sun).
  - Mary is a female. — female(mary).
  - Beethoven lived between 1770 & 1827. — person(beethoven,1770,1827).

Prolog: facts – cont'd

- **Facts about facts:**
  - Full stop "." at the end of every fact.
  - The number of arguments in a fact is called arity.
    - E.g. female(mary). is an instance of female/1 (functor female, arity 1)
  - Facts with different number of arguments are distinct
    - E.g. female(mary,may). is different from female(mary).

Prolog: rules

- **A rule in Prolog is in a full horn clause format:**

$$c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$$

- **Syntax:**

$$\underbrace{\texttt{rel}_1}_{\text{head}} \texttt{ :- } \underbrace{\texttt{rel}_2\texttt{, rel}_3\texttt{, ... rel}_n}_{\text{body}}\texttt{.}$$

  - *If I know that all those relations (those in the body) hold, then I also know that this LHS relation (in the head) holds.*

- **Examples:**
  - If there is smoke there is fire
    fire :- smoke.
  - If the course is boring, I leave
    leave(i) :- boring(course).
  - Joe is going to kill the teacher if he fails CSC324.
    kills(joe, X) :- fails(joe,csc324), teaches(X,csc324).

Prolog: rules – cont’d

- **Examples:**
  - X is female if X is the mother of anyone.
    female(X) :- mother(X,_). % avoid singleton variables by using _.
  - X is the sister of Y, if X is female and X's parents are M and F, and Y's parents are M and F
    sister_of(X,Y):- female(X),parents(X,M,F),parents(Y,M,F).
    % in general, how we interpret the rule in first-order logic (predicate logic)?
- **When to use rules?**
  - Use rules to say that a particular fact depends on a group of facts.
  - Use rules to deduce new facts from existing ones.
- **Rules of rules:**
  - The head of the rule consist of at most one predicate
  - The body of the rule is a finite sequence of literals separated by ',' (which means conjunction *and*)
  - Rules always end with a period “.”

Prolog: queries

- **A query is a clause with an empty head.**
  $\leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$
- **Syntax**
  ?- <body>.
  - Try to prove that <body> is true
  - The goal is represented to the interpreter as a question.
- **Examples**

?- round(earth). % is it true that the earth is round?
% (or simpler than that: is the earth round?)

?-round(X). % is it true that there are entities which are round?
% (or simpler than that: what entities are round?)

Prolog: queries – cont’d

- **Examples**

?- composer(beethoven,1770,1827).
- is it true that beethoven was a composer who lived between 1770 and 1827

?- owns(john,book). - is it true that john owns a book?
(simpler: does john own a book?)

?-owns(john,X). - is it true that john owns something?
(simpler: does john own something?)

Prolog: simple types - constants

- **There are two types of constants: *atoms* and *numbers*.**
- **Atoms:**
  - Alphanumeric atoms: *alphabetic sequence starting with a lower case letter*
    - E.g.: apple a1 apple_cart
  - Special atoms
    - E.g ! ; [ ]
  - Symbolic atoms: *sequence of symbolic characters*
    - E.g. & < > * - +
  - Quoted atoms: *sequence of characters surrounded by single quotes*
    - Can make anything an atom by enclosing it in single quotes.
    - E.g ‘apple’ ‘hello world’
- **Numbers:**
  - Integers and Floating Point numbers
    - E.g. 0 1 9821 -10 1.3 -1.3E102

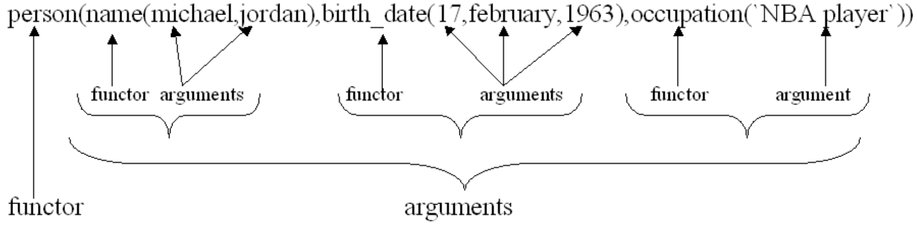
Prolog: complex types - structures

- **Recall: what's a functional term?**
  functor(some-parameters)   e.g. office(mary)

- **We can construct complex data structures using nested *functional terms*.**
  – Represents a statement about the world

- **Example:**
  – A person has; name: first name, last name - birth date: day, month, year & occupation

person(name(michael,jordan),birth_date(17,february,1963),occupation(`NBA player`))



Prolog: complex types - structures

Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query:
"Retrieve everything that John owns."
i.e. , Find `X` such that `owns(john,X)` is true.

```
answers:  X = car(red,corvette)
          X = cat(black,siamese,sylvester)
```

Query:
"Retrieve the colour and make of John's car."
i.e., `owns(john,car(Colour,Make))`

```
answer:  Colour = red
         Make = corvette
```

Prolog: complex types - structures

Same Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query: "Who owns a red car?"
i.e., Find values for `Who` so that
∃`Make owns(Who,car(red,Make))` is true.

```
answers:  Who = john
          Who = elvis
```

Prolog: complex types - structures

Same Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query: "Who owns a copyright?"
i.e., Find values for `Who` so that
∃`X,Y owns(Who,copyright(X,Y))` is true.

```
answers:  Who = elvis
          Who = tolstoy
```

Prolog: an example

Facts

```
likes(eve, pie).        food(pie).
likes(al, eve).         food(apple).
likes(eve, tom).        person(tom).
likes(eve, eve).
```

query

```
?-likes(al, pie).
no
?-likes(al,eve).
yes
?-likes(eve,al).
no
?-likes(person,food).
no
```

answer

variable

```
?-likes(al,Who).
Who=eve
?-likes(eve,W).
W=pie  ;
W=tom  ;
W=eve  ;
no
```

answer with variable binding

force search for more answers

Prolog: example – cont'd

Facts

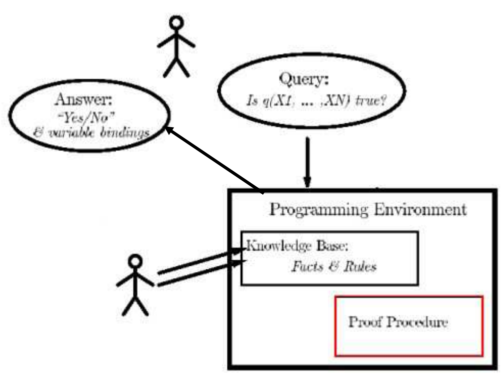
```
likes(eve, pie).        food(pie).
likes(al, eve).         food(apple).
likes(eve, tom).        person(tom).
likes(eve, eve).
```

```
?-likes(A,B).
A=eve,B=pie ; A=al,B=eve ; …
?-likes(D,D).
D=eve ; no
?-likes(eve,W), person(W).
W=tom
?-likes(al,V), likes(eve,V).
V=eve ;  no
```

and

Prolog: proof procedure

- **Two main processes:**
  - Unification
  - Top-down reasoning



Prolog: unification

- **First step in proof procedure**
- **Prolog tries to satisfy a query by *unifying* it with some conclusion and see if it is true!**
- **Process of finding these suitable "assignments" of values to variables is called *unification***
  - It is really a process of pattern matching to make statements identical
  - Somewhat similar to variable bindings in imperative world and to pattern matching in Scheme.

Prolog: unification – cont’d

- **Rules of unification:**

Object 1	Object 2	example		result
constant	free var.	4	X	X=4
bound variable	free variable	X	Y	Y gets the value of X
free variable	bound variable	X	Y	X gets the value of Y
bound variable	constant	X	“b”	fails if X has a value different then “b”
compound object	compound object	f(X,Y)	f(2,3)	X=2, Y=3
compound object	compund object	f(q(2,X),3)	f(P,3)	succeds if P is free, and P=q(2,X) . (.. more posibilities)
compund object	compound object	f(3,X)	q(3,X)	fails, due to different functors (p is not q)

Prolog: unification – cont’d

- **Rules of unification:**
  - A constant unifies only with itself, it cannot unify with any other constant.
  - Two structures unify iff they have <u>the same name</u>, <u>number of arguments</u> and <u>all the arguments unify</u>.
  - Unification requires all instances of the same variable in a rule to get the same value

Prolog: unification – cont’d

- **Examples:**

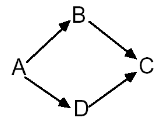
`a(b,C,d,E)`	
with `x( ... )`	doesn’t unify: `a` and `x` differ
`a(b,C,d,E)`	
`a(_,_,_)`	no: different # of args
`a(b,C,d,E)`	
`a(j,f,G,H)`	no: `b` $\neq$ `j`
`a(b,C,d,E)`	
`a(b,f,G,H)`	yes: by either $\{C \mapsto f, G \mapsto d, H \mapsto E\}$ or $\{C \mapsto f, G \mapsto d, E \mapsto H\}$
`a(pred(X,j))`	
`a(pred(k,j))`	yes: $\{X \mapsto k\}$
`a(pred(X,j))`	
`a(B)`	yes: $\{B \mapsto$ `pred(X,j)` $\}$

Prolog: unification – cont’d

- **Examples:**
  - Does p(X,X) unify with p(b,b) ?
  - Does p(X,X) unify with p(b,c) ?
  - Does p(X,b) unify with p(Y,Y) ?
  - Does p(X,Z,Z) unify with p(Y,Y,b) ?
  - Does p(X,b,X) unify with p(Y,Y,c) ?
    - To make the third arguments equal, we must replace X by c
    - To make the second argument equal, we must replace Y by b.
    - So, p(X,b,X) becomes p(c,b,c), and p(Y,Y,c) becomes p(b,b,c).
    - However, p(c,b,c) and p(b,b,c) are not syntactically identical.

Prolog: example 2

- **Facts & rules:**



link(a,b), link(b,c), link(a,d), link(d,c).
path(N, N).
path(L, M) :- link(L, X), path(X, M).

- **Posing queries:**

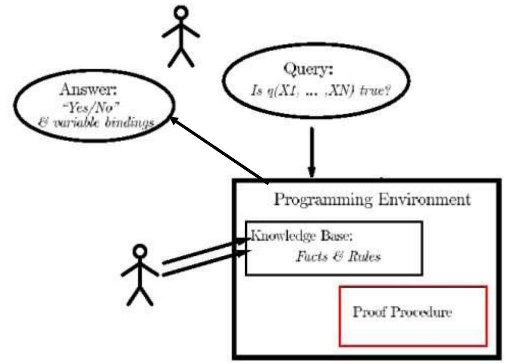
Based on our logical encoding of the graph, we can then write queries:

```
?- path(a,c)
yes

?- path(c,a)
no

?- path(a,X), path(X,c)
X = a
X = b
X = c
X = d
```

Notice that we didn't write a graph traversal algorithm, and we didn't hard code the set of questions we can ask in advance. We just define what a graph is…

Prolog: proof procedure - revisited

- **Two main processes:**
  - ✓ Unification
  - – Top-down reasoning



Prolog: reasoning

- **Given a set of facts and rules, we need a mechanism to deduce new facts and/or prove that a given rule is true or false or has no answer**

- **There are two techniques to do this:**
  - Bottom-up reasoning
  - Top-down reasoning

Bottom-up Reasoning

- Bottom-up (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

  e.g., Suppose the fact B and the rule $A \leftarrow B$ are given. Then infer that A is true.

Example

Rule base:

```
p(X,Y,Z) <- q(X),q(Y),q(Z).
q(a1).
q(a2).
...
q(an).
```

Bottom-up inference derives n^3 facts of the form p(a_i,a_j,a_k):

```
p(a1, a1, a1)
p(a1, a1, a2)
p(a1, a2, a3)
...
```

A rule base:

```
A <- B    (1)
B <- C    (2)
C         (3)
```

A bottom-up proof:

infer A

↑ rule (1)

infer B

↑ rule (2)

infer C

↑ rule (3)

start

So, A is proved

Prolog: top-down reasoning

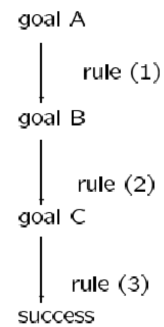
- Top-down (or backward) reasoning: starting from the query, apply the rules in reverse, attempting only those lines of inference that are relevant to the query.

  e.g., Suppose the query is A, and the rule $A \leftarrow B$ is given. Then to prove A, try to prove B.

A rule base:

```
A <- B     (1)
B <- C     (2)
C          (3)
```

A top-down proof:



So, A is proved

Prolog: top-down reasoning – cont'd

- **Multiple rules and multiple premises:**
  - A fact may be inferred by many rules
    - E.g. `E <- B`
      `E <- C`
      `E <- D`
  - A rule may have many premises
    - E.g. `E <- B /\ C /\ D`
- **In top-down inference, such rules give rise to**
  - Inference trees
  - Backtracking

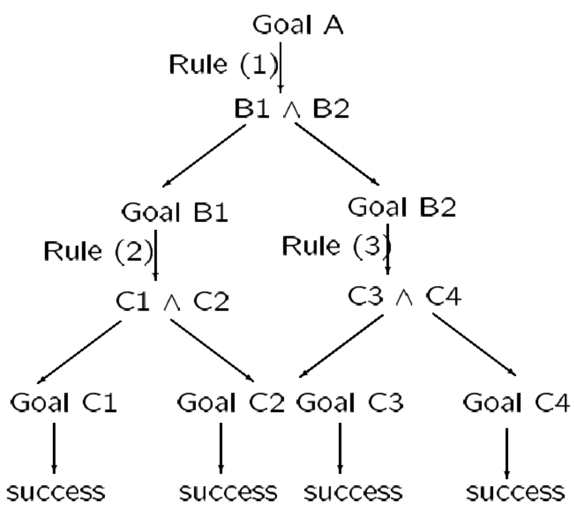
Prolog: top-down reasoning – cont'd

- **Example:** *multiple premises*

Rule base:

```
(1)   A  <- B1 /\ B2
(2)   B1 <- C1 /\ C2
(3)   B2 <- C3 /\ C4
      C1    C2    C3    C4
```

Query: Is A true?



So, goal A is proved. (all paths must succeed)

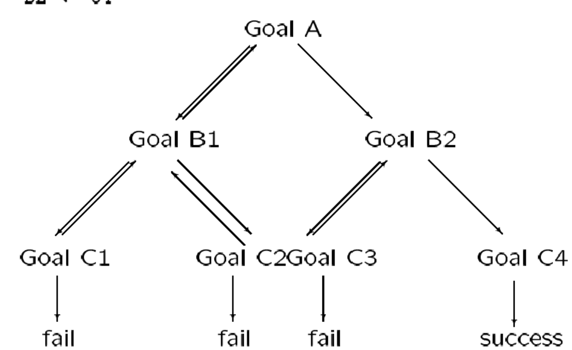
Prolog: top-down reasoning – cont'd

- **Example:** *multiple rules*

Rule base:

```
A <- B1     B1 <- C1     B2 <- C3
A <- B2     B1 <- C2     B2 <- C4
C4
```

Query: Is A true?



So, goal A is proved. (only one path must succeed

Prolog: backtracking

- **Prolog uses this algorithm for proving a goal by recursively breaking goal down into sub-goals and try to prove these sub-goals until facts are reached.**

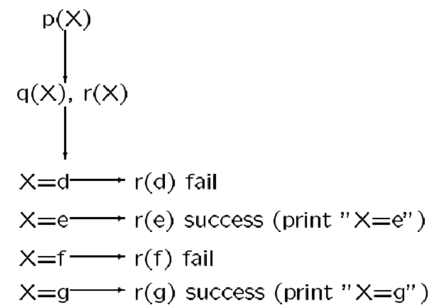
- **To satisfy a goal:**
  - Try to unify with conclusion of first rule in database
  - If successful, apply substitution to first premise, try to satisfy resulting sub-goals
  - Then apply both substitutions to next sub-goal (premise), and so on...
  - If not successful, go on to the next rule in database
  - If all rules fail,try again (backtrack) to a previous sub-goal

Prolog: backtracking example 1

Rule base:

```
p(X) :- q(X),r(X).
q(d).   q(e).   q(f).   q(g).
r(e).   r(g).
```

Query: Find `X` such that `p(X)` is true.

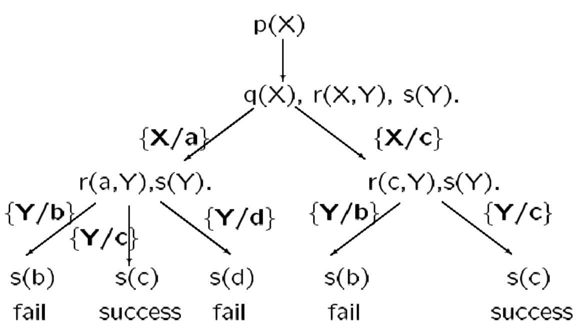


Prolog: backtracking example 2

Rule base:

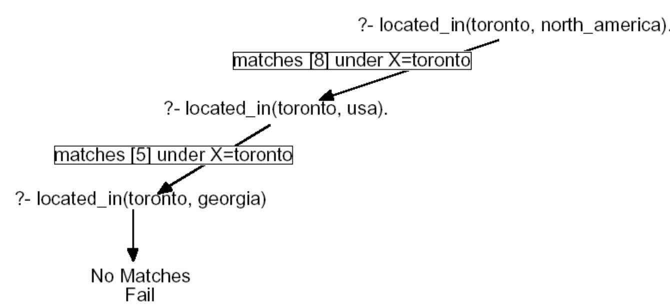
```
p(X) :- q(X),r(X,Y),s(Y).
q(a).   r(a,b).   r(c,b).   s(c).
q(c).   r(a,c).   r(c,c).
        r(a,d).
```

Query: Find `X` such that `p(X)` is true.



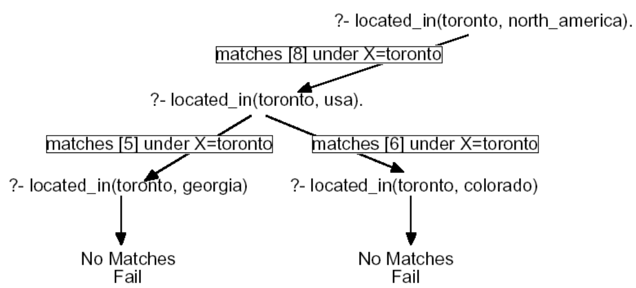
Prolog: backtracking example 3

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



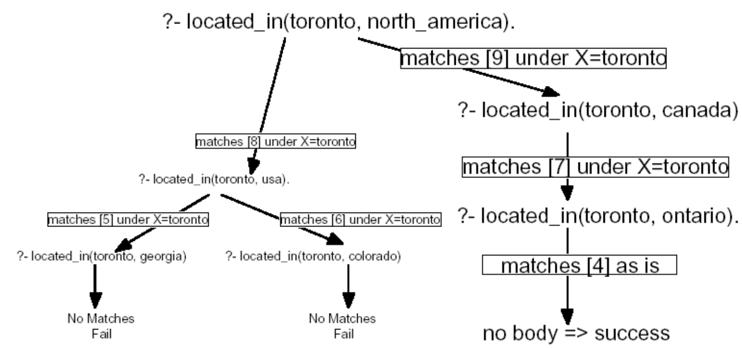
Prolog: backtracking example 3 – cont'd

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



Prolog: backtracking example 3 – cont'd

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



Top-down vs. Bottom-up Reasoning

- **Prolog uses top-down inference, although some other logic programming systems use bottom-up inference (e.g. Coral)**
- **Each has its own advantages and disadvantages:**
  - Bottom-up may generate many irrelevant facts
  - Top-down may explore many lines of reasoning that fail.
- **Top-down and bottom-up inference are logically equivalent**
  - i.e. they both prove the same set of facts.
- **However, only top-down inference simulates program execution**
  - i.e. execution is inherently top down, since it proceeds from the main procedure downwards, to subroutines, to sub-subroutines, etc...