CSC384: Intro to Artificial Intelligence

- Brief Introduction to Prolog
  - Part 1/2: Basic material
  - Part 2/2 : More advanced material

Hojjat Ghaderi and Fahiem Bacchus, University of Toronto

CSC384: Intro to Artificial Intelligence

- Resources
  - Check the course website for several online tutorials and examples.
  - There is also a comprehensive textbook: *Prolog Programming for Artificial Intelligence by Ivan Bratko*.

What's Prolog?

- Prolog is a language that is useful for doing symbolic and logic-based computation.
- It's *declarative*: very different from imperative style programming like Java, C++, Python,…
- A program is partly like a database but much more powerful since we can also have general **rules** to infer new facts!
- A **Prolog interpreter** can follow these facts/rules and answer **queries** by sophisticated **search**.
- *Ready for a quick ride? Buckle up!*

What's Prolog? Let's do a test drive!

Here is a simple Prolog program saved in a file named ***family.pl***

```
male(albert).                    %a fact stating albert is a male
male(edward).
female(alice).                   %a fact stating alice is a female
female(victoria).
parent(albert,edward).           %a fact: albert is parent of edward
parent(victoria,edward).
father(X,Y) :-                   %a rule: X is father of Y if X if a male parent of Y
        parent(X,Y), male(X). %body of above rule, can be on same line.
mother(X,Y) :- parent(X,Y), female(X).  %a similar rule for X being mother of Y
```

- A fact/rule (statement) ends with "." and white space ignored
- read :- after rule head as "if". Read comma in body as "and"
- Comment a line with % or use /* */ for multi-line comments
- Ok, how do we run this program? What does it do?

Running Prolog

- There are many Prolog interpreters.
- **SWI** is the one we use on **CDF**. Type **prolog** in command line:

```
skywolf:~% prolog
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.58)
Copyright (c) 1990-2008 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).

?-   ← this is the prompt. You can load you program and ask queries.

?- consult(family).    %loading our file. We can use full-name with quotation 'family.pl'

% family compiled 0.00 sec, 2,552 bytes    %this is the result of loading
true.                                       %true means loading file was successful

                % alternatively, we could have used ['family.pl']. to load our file.

?- halt.                %exiting SWI!
skywolf:~%
```

But what can we do with our program?

We can ask **queries** *after loading our program*:

?-male(albert).
Yes. *%the above was true*
?-male(victoria).
No. *%the above was false*
?-male(mycat).
No.
?-male(X). *%X is a **variable**, we are asking "**who** is male?"*
X=albert; *%right! now type **semicolon** to ask for more answers.*
X=edward;
No *%no more answers*
?-father(F,C). *%F & C are variables, we are asking "**who** is father of **whom**"*
F=albert, C=edward; ***%this is awesome! How did Prolog get this?***
No

```
male(albert).     %this is our family.pl program
male(edward).
female(alice).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
father(X,Y):- parent(X,Y), male(X).
mother(X,Y):- parent(X,Y), female(X).
```

What to learn?

- Your first impression? It's very different from Java, Python, and C!
- We first need to learn the **Prolog Syntax**, similar to learning any other language!
- To learn how to write programs and ask queries, we also need to understand **how a Prolog interpreter operates** to find answers to our queries.
- Finally, you will need to learn how to write more efficient programs, how to use Negation as Failure, and how to control the Prolog search mechanism using cut.

Syntax of Prolog

- Terms
- Predicates
- Facts and Rules
- Programs
- Queries

Syntax of Prolog: Terms

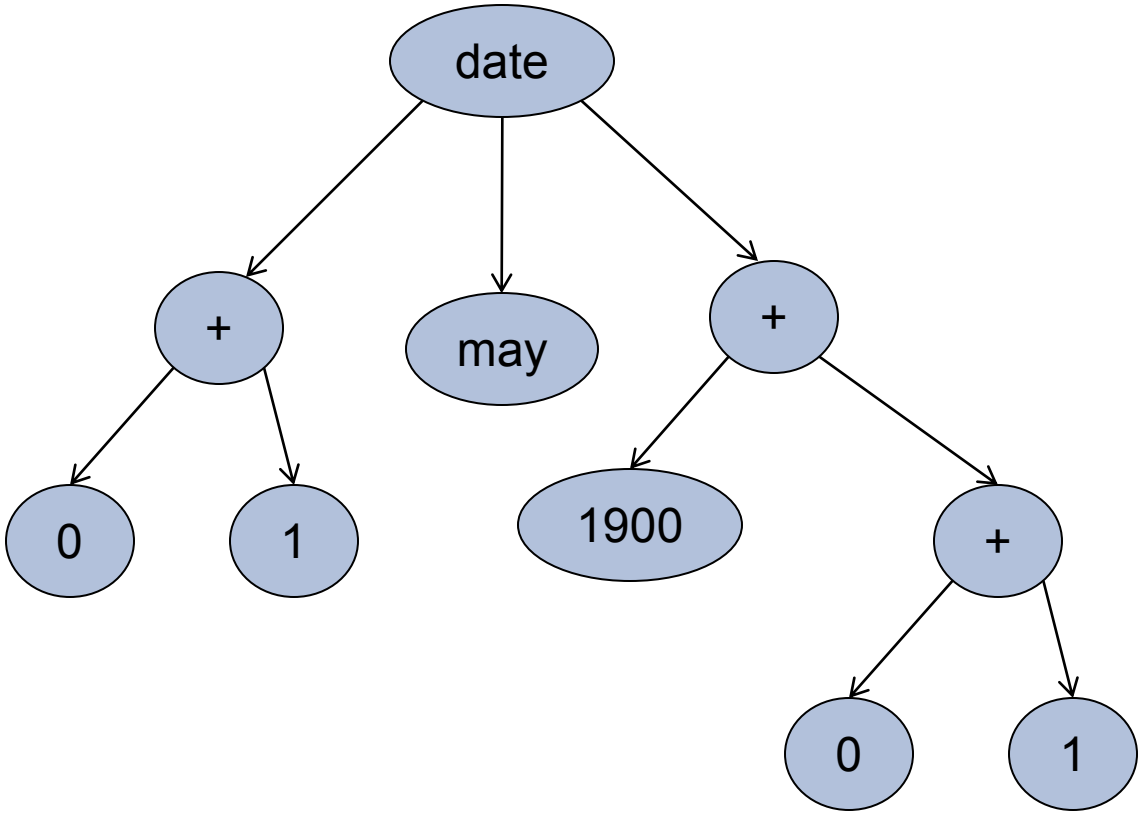
- Constants:
  - Identifiers
    - sequences of letters, digits, or underscore "_" that start with lower case letters.
    - mary, anna, x25, x_25, alpha_beta
  - Numbers
    - 1.001, 2, 3.03
  - Strings enclosed in single quotes
    - 'Mary', '1.01', 'string'
      - Note can start with upper case letter, or can be a number now treated as a string.
- Variables
  - Sequence of letters digits or underscore that start with an upper case letter or the underscore.
    - _x, Anna, Successor_State,
    - Undescore by itself is the special "anonymous" variable.

Syntax of Prolog: Terms

- Constants:
- Variables
- Structures (like function applications)
  - <identifier>(Term1, ..., Termk)
    - date(1, may, 1983)
    - point(X, Y, Z)
  - Note that the definition is recursive. So each term can itself be a structure
    - date(+(0,1), may, +(1900,-(183,100)))
  - Structures can be represented as a tree

Syntax of Prolog: Terms

- Structures as trees
  - date(+(0,1), may, +(1900,-(183,100)))



Syntax of Prolog: Terms

- Structures
  - Rather than represent the arithmetic term
    +(1900,-(183,100))
    in this format (prefix form) Prolog will represent it in more standard infix form
    1900 + (183 – 100)
  - Note that since Prolog is a symbolic language it will treat this arithmetic expression as a symbol. That is, it will not evaluate this expression to 1983.
  - To force the evaluation we use "is"
    X is 1900 + (183 – 100).

Syntax of Prolog: Lists as special terms

- Lists are a very useful data structure in Prolog.
- Lists are structured terms represented in a special way.
  - [a,b,c,d]
    - This is actually the structured term
      [a | [c | [b | [d | []]]]]
    - Where [] is a special constant the empty list.
    - Each list is thus of the form [<head> | <rest_of_list>]
    - <head> an element of the list (not necessarily a list itself).
    - <rest_of_list> is a list (a sub-list).
    - also, [a,b,c,d]=[a | [b,c,d]] =[a,b | [c,d]] = [a,b,c | [d]]
    - List elements can be any term! For example the list
      [a, f(a), 2, 3+5, point(X,1.5,Z)] contains 5 elements.
  - As we will see, this structure has important implications when it comes to matching variables against lists!

Syntax of Prolog: Predicates

- Predicates are syntactically identical to structured terms

<identifier>(Term1, ..., Termk)

  - elephant(mary)
  - taller_than(john, fred)

Syntax of Prolog: Facts and Rules

- A prolog program consists of a collection of facts and rules.
- A fact is a predicate terminated by a period "."

<identifier>(Term1, ..., Termk).

- Facts make assertions:
  - elephant(mary). — Mary is an elephant.
  - taller_than(john, fred). — John is taller than Fred.
  - parent(X). — Everyone is a parent!
    - Note that X is a variable. X can take on any term as its value so this fact asserts that for every value of X, "parent" is true.

Syntax of Prolog: Facts and Rules

- Rules

  predicateH :- predicate1, ..., predicatek.

- First predicate is RULE HEAD. Terminated by a period.
- Rules encode ways of deriving or computing a new fact.
  - animal(X) :- elephant(X).
    - We can show that X is an animal if we can show that it is an elephant.
  - taller_than(X,Y) :- height(X,H1), height(Y,H2), H1 > H2.
    - We can show that X is taller than Y if we can show that H1 is the height of X, and H2 is the height of Y, and H1 is greater than H2.
  - taller_than(X,Jane) :- height(X,H1), H1 > 165
    - We can show that X is taller than Jane if we can show that H1 is the height of X and that H1 is greater than 165
  - father(X,Y) :- parent(X,Y), male(X).
    - We can show that X is a father of Y if we can show that X is a parent of Y and that X is male.

Operation Of Prolog

- A query is a sequence of predicates
  - predicate1, predicate2, ..., predicatek
- Prolog tries to prove that this sequence of predicates is true using the facts and rules in the Prolog Program.
- In proving the sequence it performs the computation you want.

Example

```
elephant(fred).
elephant(mary).
elephant(joe).
animal(fred) :- elephant(fred).
animal(mary) :- elephant(mary).
animal(joe) :- elephant(joe).
```

QUERY

```
animal(fred), animal(mary), animal(joe)
```

Operation

Starting with the first predicate P1 of the query
Prolog examines the program from **TOP** to **BOTTOM**.
It finds the first RULE HEAD or FACT that matches P1
Then it replaces P1 with the RULE BODY.
If P1 matched a FACT, we can think of FACTs as having empty bodies (so P1 is simply removed).
The result is a new query.

E.g.
P1 :- Q1, Q2, Q3

QUERY = P1, P2, P3

P1 matches with rule
New QUERY = Q1, Q2, Q3, P2, P3

Example

```
elephant(fred).
elephant(mary).
elephant(joe).
animal(fred) :- elephant(fred).
animal(mary) :- elephant(mary).
animal(joe) :- elephant(joe).
```

QUERY

```
animal(fred), animal(mary), animal(joe)
```

1. elephant(fred), animal(mary), animal(joe)
2. animal(mary),animal(joe)
3. elephant(mary), animal(joe)
4. animal(joe)
5. elephant(joe)
6. EMPTY QUERY

Operation

- If this process reduces the query to the empty query, Prolog returns "yes".
- However, during this process each predicate in the query might match more than one fact or rule head.
  - Prolog always choose the first match it finds. Then if the resulting query reduction did not succeed (i.e., we hit a predicate in the query that does not match any rule head of fact), Prolog backtracks and tries a new match.

Example

```
ant_eater(fred).
animal(fred) :- elephant(fred).
animal(fred) :- ant_eater(fred).
```

QUERY

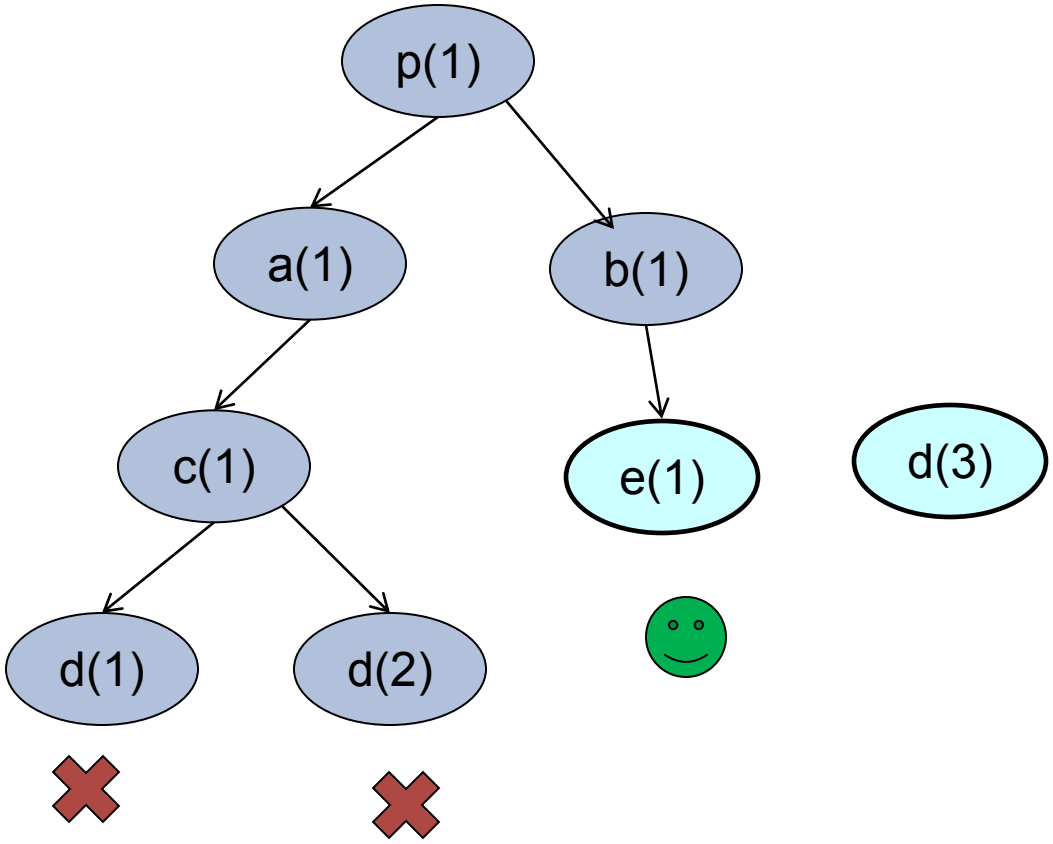
```
animal(fred)
```

1. elephant(fred).
2. FAIL BACKTRACK.
3. ant_eater(fred).
4. EMPTY QUERY

Operation

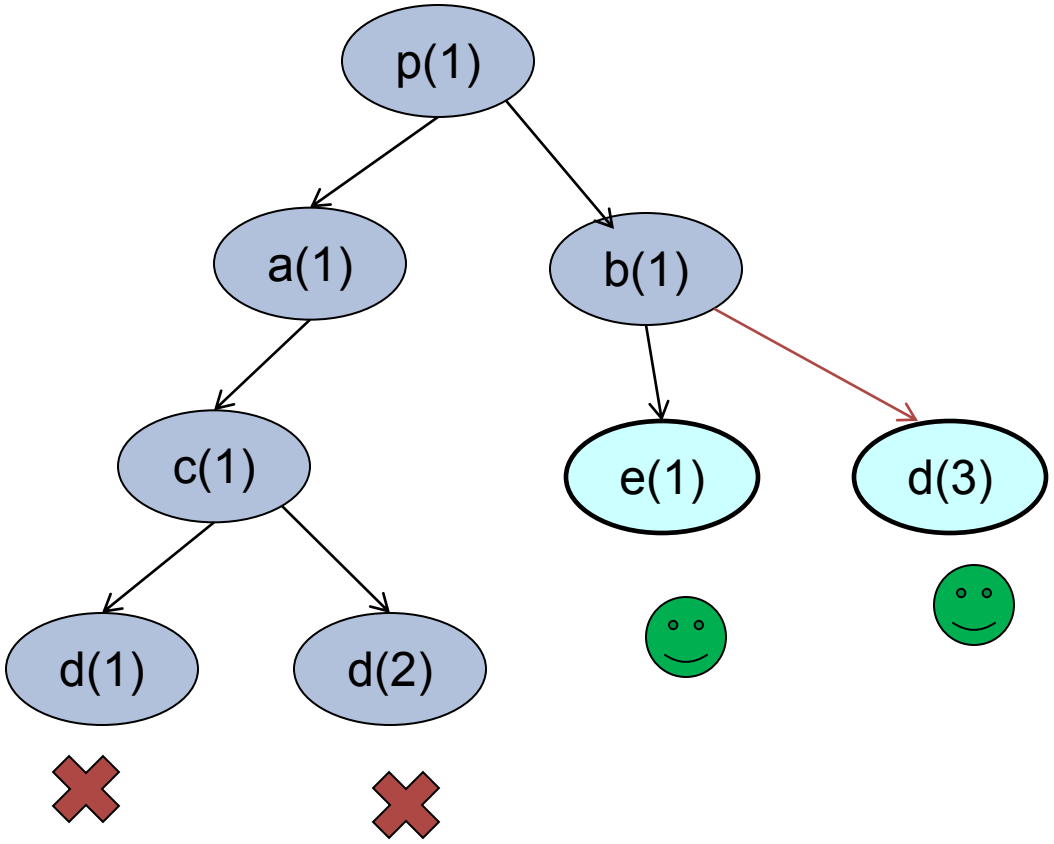
- Backtracking can occur at every stage as the query is processed.

- p(1) :- a(1).
  p(1) :- b(1).
  a(1) :- c(1).
  c(1) :- d(1).
  c(1) :- d(2).
  b(1) :- e(1).
  e(1).
  d(3).
- **Query**: p(1)



Operation

- With backtracking we can get more answers by using ";"
- p(1) :- a(1).
  p(1) :- b(1).
  a(1) :- c(1).
  c(1) :- d(1).
  c(1) :- d(2).
  b(1) :- e(1).
  b(1) :- d(3).
  e(1).
  d(3).
- **Query**: p(1)



Variables and Matching

- Variables allow us to
  - Compute more than yes/no answers
  - Compress the program.
  - elephant(fred).
    elephant(mary).
    elephant(joe).
    animal(fred) :- elephant(fred).
    animal(mary) :- elephant(mary).
    animal(joe) :- elephant(joe).

  - The three rules can be replaced by the single rule
    animal(X) :- elephant(X).
  - When matching queries against rule heads (of facts) variables allow many additional matches.

Example

```
elephant(fred).
elephant(mary).
elephant(joe).
animal(X) :- elephant(X).
```

QUERY

animal(fred), animal(mary), animal(joe)

1. X=fred, elephant(X), animal(mary), animal(joe)
2. animal(mary),animal(joe)
3. X = mary, elephant(X), animal(joe)
4. animal(joe)
5. X= joe, elephant(X)
6. EMPTY QUERY

Operation with Variables

- Queries are processed as before (via rule and fact matching and backtracking), but now we can use variables to help us match rule heads or facts.
- A query predicate matches a rule head or fact (either one with variables) if
  - The predicate name much match. So elephant(X) can match elephant(fred), but can never match ant_eater(fred).
  - Once the predicates names the arity of the predicates much match (number of terms). So foo(X,Y) can match foo(ann,mary), but cannot match foo(ann) or foo(ann,mary,fred).

Operation

- A query predicate matches a rule head or fact (either one with variables) if
  - If the predicate names and arities match then each of the k-terms much match. So for foo(t1, t2, t3) to match foo(s1, s2, s3) we must have that t1 matches s1, t2 matches s2, and t3 matches t3.
  - During this matching process we might have to "bind" some of the variables to make the terms match.
  - These bindings are then passed on into the new query (consisting of the rule body and the left over query predicates).

Variable Matching (Unification)

- Two terms (with variables match if):
  - If both are constants (identifiers, numbers, or strings) and are identical.
  - If one or both are **bound** variables then they match if what the variables are bound to match.
    - X and mary where X is bound to the value mary will **match**.
    - X and Y where X is bound to mary and Y is bound to mary will **match**,
    - X and ann where X is bound to mary will not match.

Variable Matching (Unification)

- If one of the terms is an **unbound** variable then they match **AND** we bind the variable to the term.
  - X and mary where X is unbound match and make X bound to mary.
  - X and Y where X is unbound and Y is bound to mary match and make X bound to mary.
  - X and Y where both X and Y are unbound match and make X bound to Y (or vice versa).

Variable Matching (Unification)

- If the two terms are structures
  t = f(t1, t2, ..., tk)
  s = g(s1, s2, ..., sn)
- Then these two terms match if
  - the identifiers "f" and "g" are identical.
  - They both have identical arity (k=n)
  - Each of the terms ti, si match (recursively).
- E.g.
  - date(X, may, 1900) and date(3, may, Y) match and make X bound to 3 and Y bound to 1900.
  - equal(2 + 2, 3 + 1) and equal(X + Y, Z) match and make X bound to 1, Y bound to 2, and Z bound to "3+1".
    - Note that to then evaluate Z by using "is".
  - date(f(X,a), may, g(a,b)) and date(Z, may, g(Y,Q)) match and make Z bound to "f(X,a)", Y bound to a, and Q bound to b.
    - Note we can bind a variable to a term containing another variable!
- The predicate "=" shows what Prolog unifies!

Unification Examples

- Which of the following are unifiable?

Term1	Term 2	Bindings if unifiable
X	f(a,b)	X=f(a,b)
f(X,a)	g(X,a)	
3	2+1	No! use is if want 3
book(X,1)	book(Z)	
[1,2,3]	[X | Y]	X=1, Y=[2,3]
[a,b,X]	[Y | [3,4]]	
[a | X]	[X | Y]	X=a Y=a improper list
X(a,b)	f(Z,Y)	
[X | Y | Z]	[a,b,c,d]	X=a. Y=b, Z=[c,d]

Solving Queries

- How Prolog works:
  - Unification
  - Goal-Directed Reasoning
  - Rule-Ordering
  - DFS and backtracking

- When given a query Q = q1, q2, …, qn Prolog performs a search in an attempt to solve this query. The search can be specified as follows

Details of Solving Queries by Prolog

```
//note:  variable bindings are global to the procedure "evaluate"
bool evaluate(Query Q)
  if Q is empty
    SUCCEED: print bindings of all variables in original query
       if user wants more solutions (i.e. enters ';') return FALSE
       else return TRUE
  else
    remove first predicate from Q, let q1 be this predicate
    for each rule R = h :- r1, r2, ..., rj in the program in
                  the order they appear in the file
      if(h unifies with q1 (i.e., the head of R unifies with q1))
        put each ri into the query Q so that if the Q was originally
        (q1, q2, ..., qk) it now becomes (r1, r2, ..., rj, q2, ... qk)
              NOTE: rule's body is put in front of previous query predicates.
              NOTE: also some of the variables in the ri's and q2...qk might
                now be bound because of unifying h with q1
         if (evaluate(Q) )   //recursive call on updated Q
            return.  //succeeded and printed bindings in recusive call.
```

Computing with Queries

for each rule R = h :- r1, r2, ..., rj in the program in
the order the rules appear in the prolog file
if(h unifies with q1 (i.e., the head of R unifies with q1))
put each ri into the query Q so that if the Q was originally (q1, q2, …, qk) it now becomes (r1, r2, …, rj, q2, … qk)
NOTE: rule's body is put in front of previous query predicates.
NOTE: also some of the variables in the ri's and q2…qk might now be bound because of unifying h with q1
if(evaluate(Q))
return. //succeeded and printed bindings in recusive call.
else
UNDO all changes to the variable bindings that arose from unifying h and q1

end for
//NOTE. If R's head fails to unify with q1 we move on to try the next rule in the program. If R's head did unify but unable to solve the new query recursively, we also move on to try the next rule after first undoing the variable bindings.

Computing with Queries

```
    end for
        //NOTE. If R's head fails to unify with q1 we move on to try the
                next rule in the program. If R's head did unify but unable
                to solve the new query recursively, we also move on to
                try the next rule after first undoing the variable bindings.
    return FALSE
        //at this point we cannot solve q1, so we fail. This failure will
          unroll the recursion and a higher level recursion with then try
          different rule for the predicate it is working on.
```

Query Answering is Depth First Search

- This procedure operates like a depth-first search, where the order of the search depends on **the order of the rules** and predicates in the rule bodies.

- When Prolog backtracks, it undoes the bindings it did before.

- Exercise: try the following queries on family.pl:

  parent(P,C)

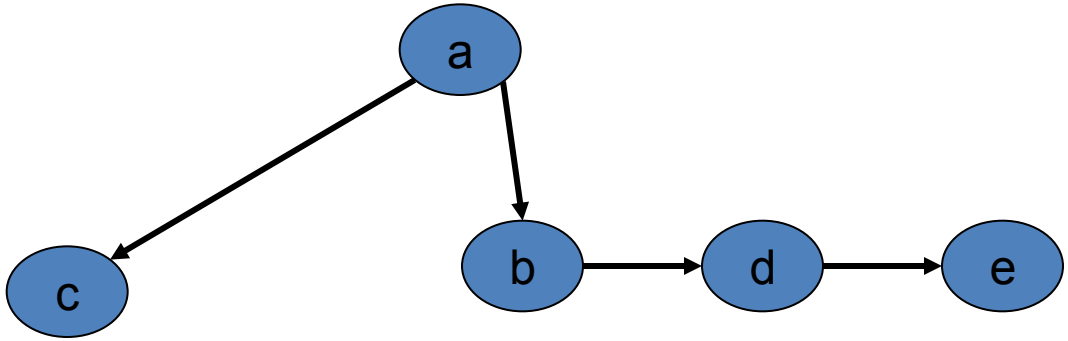
  father(F,C).

[if you can, use the interactive SWI with graphical debugger]

Another Example of Query Answering

- Route finding in in a directed acyclic graph:

```
edge(a,b).
edge(a,c).
edge(b,d).
edge(d,e).
```




```
path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

- The above is problematic. Why?
- Here is the correct solution:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Search Tree for Path Predicate

Here is the program:

```
edge(a,b).
edge(a,c).
edge(b,d).
edge(d,e).

path(X,Y) :- edge(X,Y).
path(X,Y) :-  edge(X,Z), path(Z,Y).
```

Queries:

```
path(a,e).
path(c,R).
path(S,c).
```

[run SWI interactively to see this in action.]

What if the graph is undirected? Simple:

```
undirEdge(X,Y):- edge(X,Y).
undirEdge(X,Y):- edge(Y,X).
```

then replace *edge* predicate by *undirEdge* in the definition of path.

Notes on Prolog Variables

- Prolog variables do not operate like other programming languages! You cannot change the value of variables, once they are bound to a value they remain bound. However:
  - If a variable binding contains another variable that other variable can be bound thus altering the original variable's value, e.g.

    X = f(a, g(Y)), Y = b ➔ X is bound to f(a, g(b)); Y is bound to b
  - Final answers can be computed by passing the variable's value on to a new variable. E.g., X = a, Y = b, Z = [X,Y] ➔ X is bound to a, Y is bound to b, and Z is bound to [a,b].

List Processing in Prolog

- Much of prolog's computation is organized around lists. Two key things we do with a list is iterate over them and build new ones.

- E.g. checking membership: member(X,Y) X is a member of list Y.

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

What if we define member like this:

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

what is the result of member(X,[a,b,c,d])?

- E.g. building a list of integers in range [i, j].
  build(from, to, NewList)

```
build (I,J,[ ]) :- I > J.
build (I,J,[I | Rest]) :- I =< J, N is I + 1,
                          build(N,J,Rest).
```

List Processing in Prolog continue…

- Computing the size of a list: size(List, ListSize)

```
size([],0).
size([ _|T],N) :- size(T,N1), N is N1+1.
```

- Computing the sum of a list of nums: sumlist(List, Sum)

```
sumlist([],0)
sumlist([H|T],N) :- sumlist(T,N1), N is N1+H.
```

- Exercise: implement append(L1,L2, L3) which holds if L3 is the result of appending list L1 and L2. For example, append([a,b,c],[1,2,3,4],[a,b,c,1,2,3,4]).

Next Tutorial (Part 2)

- More advanced material in the next tutorial:
  - Efficient lists processing using accumulators
  - Constructing predicates dynamically (on-the-fly)
  - Cut (controlling how Prolog does the search)
  - Negation as failure (NAF)
  - if-then-else
  - Debugging Prolog programs