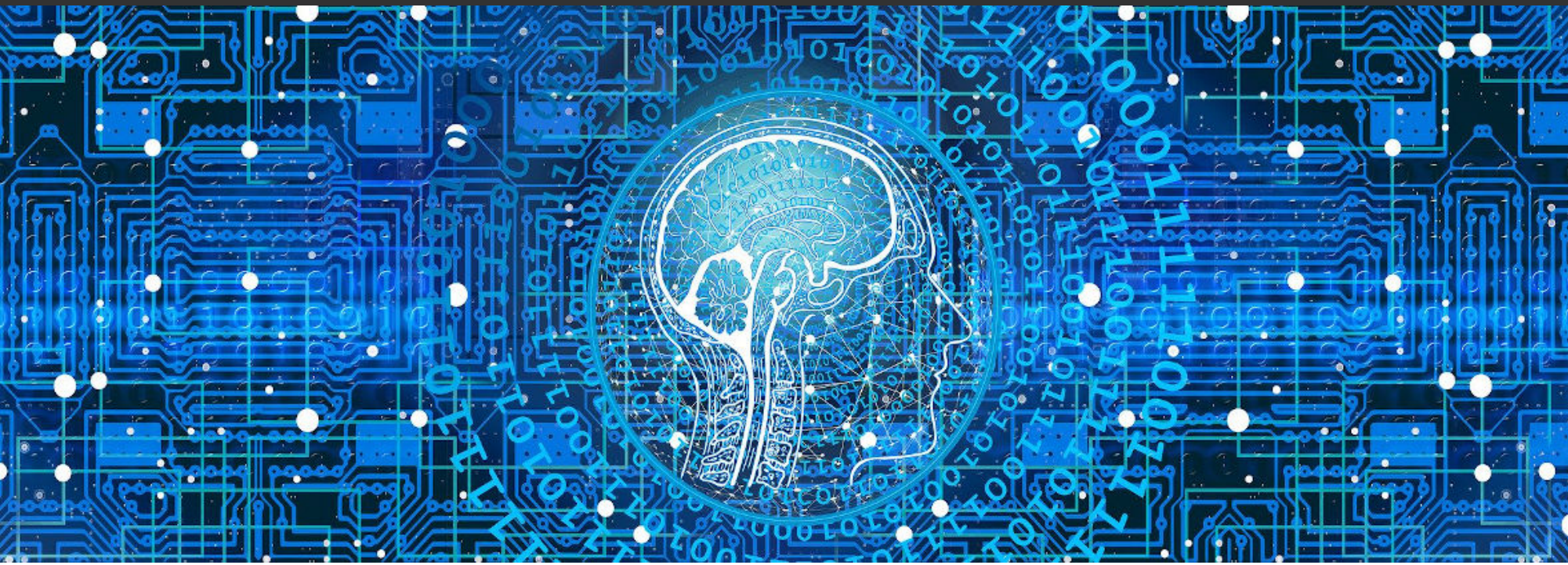


Chapter 11: Prolog

Source A: CSC384: Intro to Artificial Intelligence, Hojjat Ghaderi and Fahiem Bacchus, University of Toronto

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.

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

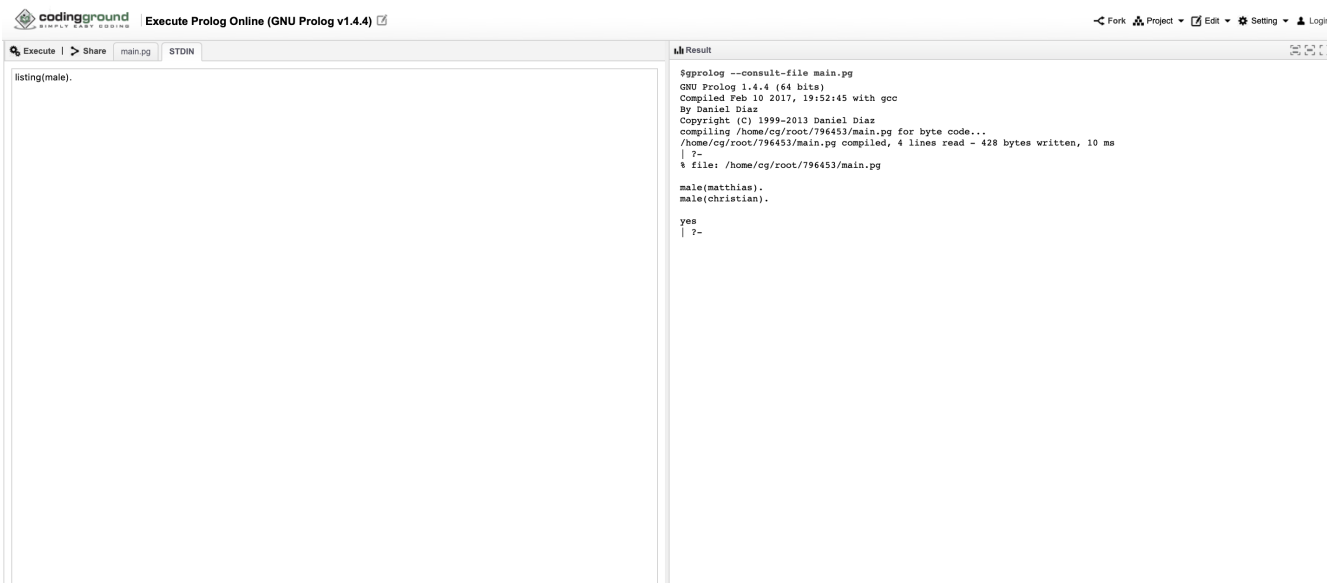
Here is a simple Prolog program saved in a file named family.pl

```
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

We can use the following Web-Version of Prolog (but you can also find Prolog-systems for your local machine)
https://www.tutorialspoint.com/execute_prolog_online.php



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.

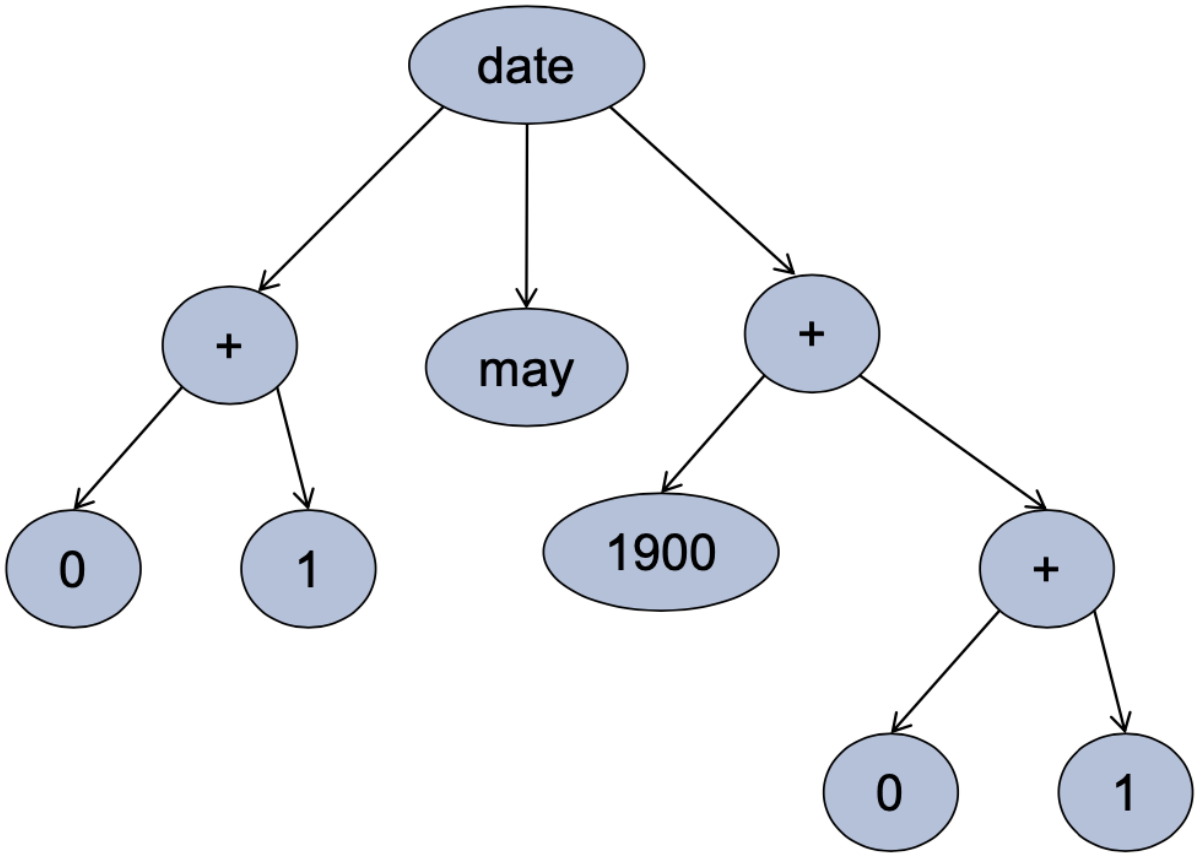
Syntax of Prolog: Terms

Structures (like function applications)

- 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.

- syntax [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

$$<\text{identifier}>(Term_1, ..., Term_k)$$

- 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 ".".

- 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 :- predicate$_1$, ..., predicate$_k$.

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
  - predicate$_1$, predicate$_2$, ..., predicate$_k$
- 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.

Example

- 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)

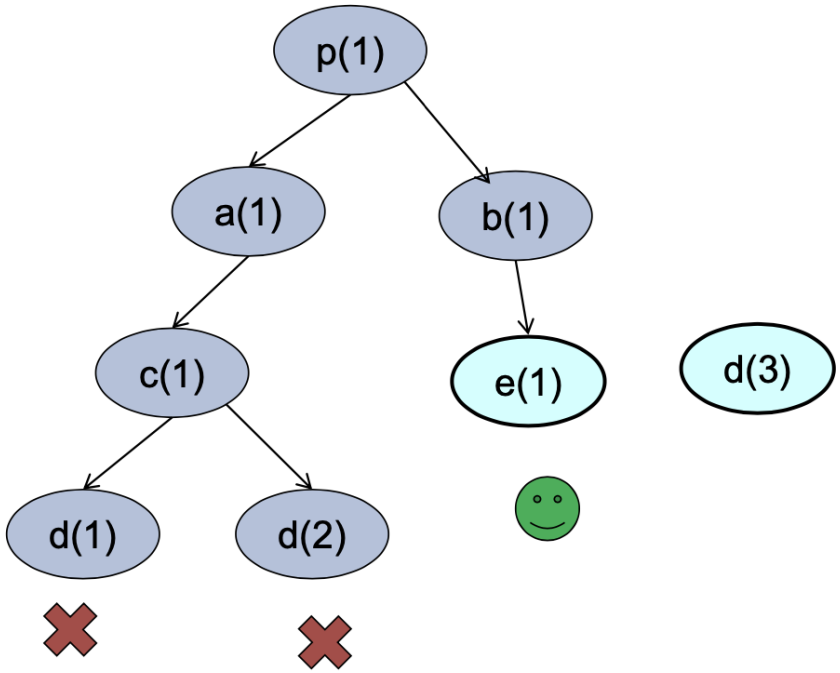
- 1. elephant(fred).
- 2. FAIL BACKTRACK.
- 3. ant_eater(fred).
- 4. EMPTY QUERY

Operation

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

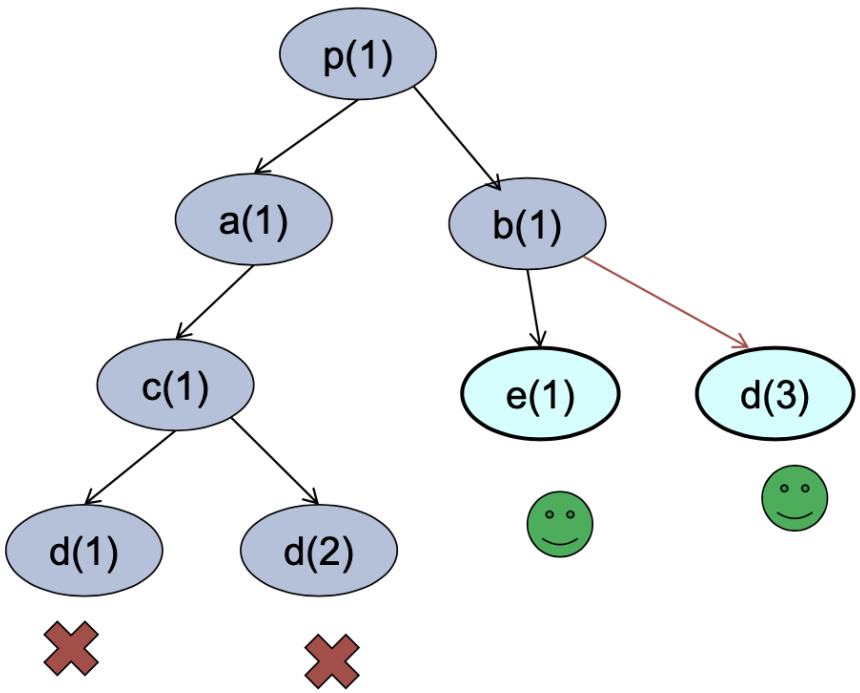
```
p(1) :- a(1).
p(1) :- b(1).
a(1) :- c(1).
c(1) :- d(1).
c(1) :- d(2).
b(1) :- e(1).
e(1).
d(3).
```

- **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($t_1, t_2, ..., t_k$)
  - s = g($s_1, s_2, ..., s_n$)
- Then these two terms match if
  - the identifiers 'f' and 'g' are identical.
  - They both have identical arity (k=n)
  - Each of the terms t_i, s_i 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 $Q = q_1, q_2, ..., q_n$ 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 (1/2)

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 (2/2)

```
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.
```

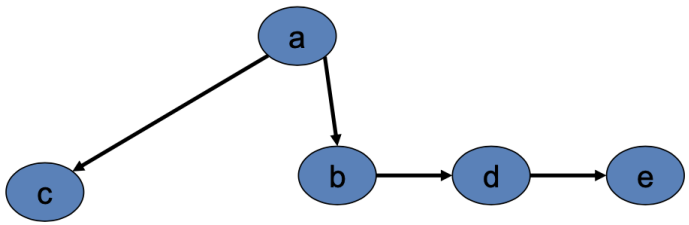
Query Answering is Depth First Search

- 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).

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).

Example 1

Knowledge Base KB1

- woman(mia).
- woman(jody).
- woman(yolanda).
- playsAirGuitar(jody).
- party.

Some Queries

- ?- woman(mia).
  - Prolog will answer: yes
- ?- playsAirGuitar(mia).
  - We will get the answer: no
- ?- playsAirGuitar(vincent).
  - Again Prolog answers no. Why? Well, this query is about a person (Vincent) that it has no information about, so it (correctly) concludes that playsAirGuitar(vincent) cannot be deduced from the information in KB1.

Example 2 - Part 1

Knowledge Base KB2

- happy(yolanda).
- listens2Music(mia).
- listens2Music(yolanda):- happy(yolanda).
- playsAirGuitar(mia):- listens2Music(mia).
- playsAirGuitar(yolanda):- listens2Music(yolanda).

Some Queries

- ?- playsAirGuitar(mia).
  - Prolog will respond yes. Why? Well, although it can't find playsAirGuitar(mia) as a fact explicitly recorded in KB2, it can find the rule
  - playsAirGuitar(mia):- listens2Music(mia).
  - Moreover, KB2 also contains the fact listens2Music(mia) . Hence Prolog can use the rule of modus ponens to deduce that playsAirGuitar(mia) .

Example 2 - Part 2

Knowledge Base KB2

- happy(yolanda).
- listens2Music(mia).
- listens2Music(yolanda):- happy(yolanda).
- playsAirGuitar(mia):- listens2Music(mia).
- playsAirGuitar(yolanda):- listens2Music(yolanda).

Some Queries

- ?- playsAirGuitar(yolanda).
  - Prolog would respond yes. Why? Well, first of all, by using the fact happy(yolanda) and the rule
  - listens2Music(yolanda):- happy(yolanda).
  - Prolog can deduce the new fact listens2Music(yolanda) . This new fact is not explicitly recorded in the knowledge base — it is only implicitly present (it is inferred knowledge). Nonetheless, Prolog can then use it just like an explicitly recorded fact.

Example 3

KB3, our third knowledge base, consists of five clauses: There are two facts, happy(vincent) and listens2Music(butch) , and three rules.

- happy(vincent).
- listens2Music(butch).
- playsAirGuitar(vincent):- listens2Music(vincent), happy(vincent).
- playsAirGuitar(butch):- happy(butch).
- playsAirGuitar(butch):- listens2Music(butch).

We see for example that Vincent plays air guitar if he listens to music and he is happy.
Some Queries

- ?- playsAirGuitar(vincent).
  - Prolog would answer no. This is because while KB3 contains happy(vincent) , it does not explicitly contain the information listens2Music(vincent) , and this fact cannot be deduced either. So KB3 only fulfils one of the two preconditions needed to establish playsAirGuitar(vincent) , and our query fails.

Example 4 - Part 1

KB4 contains only facts

- woman(mia).
- woman(jody).
- woman(yolanda).
- loves(vincent,mia).
- loves(marsellus,mia).
- loves(pumpkin,honey_bunny).
- loves(honey_bunny,pumpkin).

We see for example that Vincent plays air guitar if he listens to music and he is happy.
Some Queries

- ?- woman(X).
  - Now the first item in the knowledge base is woman(mia) . So, Prolog unifies X with mia. Prolog then reports back to us as follows: X = mia

Example 4 - Part 2

Some further Queries

- The whole point of variables is that they can stand for, or unify with, different things. And there is information about other women in the knowledge base. We can access this information by typing a semicolon:
  - X = mia ;
  - Remember that ; means or , so this query means: are there any alternatives ? So Prolog begins working through the knowledge base again (it remembers where it got up to last time and starts from there) and sees that if it unifies X with jody , then the query agrees perfectly with the second entry in the knowledge base. So it responds:
  - X = mia ;
  - X = jody
- ?- loves(marsellus,X), woman(X).
  - Now, remember that , means and , so this query says: is there any individual X such that Marsellus loves X and X is a woman
  - X = mia

Example 5

KB5 contains

- loves(vincent,mia).
- loves(marsellus,mia).
- loves(pumpkin,honey_bunny).
- loves(honey_bunny,pumpkin).
- jealous(X,Y):- loves(X,Z), loves(Y,Z).

It says that an individual X will be jealous of an individual Y if there is some individual Z that X loves, and Y loves that same individual Z too. Some Queries

- ?- jealous(marsellus,W).
  - This query asks: can you find an individual W such that Marsellus is jealous of W ? Vincent is such an individual. If you check the definition of jealousy, you'll see that Marsellus must be jealous of Vincent, because they both love the same woman, namely Mia. So Prolog will return the value
  - W = vincent