declarative/logic programming

- idea: write a program that is a logical theory about some domain and then query it
- most well known instance is Prolog
- core constructs, terms and statements, are inherited from first order logic

terms

- Prolog statements express relationships among terms
- terms are (a generalization) of the same notion in first order logic, i.e. a constant, a variable, or a function applied to some argument terms
- E.g. john, john_smith, X, Node, _person, fatherOf(paul), date(25,10,2005)
- fatherOf and date are functors; date has arity 3; it takes 3 arguments
facts

- facts are like atomic formulas in first order logic.
- syntax is same as terms, but ending with a period.
- e.g. fatherOf(paul, henry).
  mortal(ulyssus).
  likes(X, iceCream).
  likes(mary, brotherOf(helen)).
- variables are implicitly universally quantified.

rules

- rules are conditional statements.
- e.g. mortal(X) :- human(X).
  i.e. ∀x Human(x) → Mortal(x),
  all humans are mortal.
- daughter(X, Y) :- father(Y, X), female(X).
- , represents conjunction.
- likes(mary, X) :- isSweet(X).

queries

- A query asks whether a given statement is true, i.e. whether it follows from the program.
- e.g. ?- mortal(ulyssus). given
  mortal(X) :- human(X).
  human(ulyssus).
  human(penelope).
  god(zeus).
  Prolog answers Yes
queries

- `?- mortal(X).`  
  `X = ulyssus ;`  
  `X = penelope`  
  Yes

- Variables in queries are existentially quantified; can be used to retrieve information.

- Can have conjunctive queries, e.g. `?- mortal(X), mortal(Y), not(X = Y)`.

lists

- Lists are a special kind of term that allows arbitrary number of components.
  - `[]` is the empty list.
  - `.a, b)` is a dotted pair.
  - `[a, b, c] = .(a, .(b, .(c, [])))` is a list of 3 components.

- The functor `.` builds binary trees.

- Can use `display(X)` to print internal representation of `X`.

unification

- This was an instance of the kind of pattern matching called unification that Prolog performs.

- Prolog tries to find a way to instantiate the variables (substitute terms for them) that satisfies the query.

- More on this later.

- Can refer to the first and rest of a list using the notation: `[First | Rest]`

- E.g. `?- X = [a, b, c], X = [F|R].`  
  `X = [a, b, c]`  
  `F = a`  
  `R = [b, c]`  

- E.g. `X = [b], Y = a, Z = [Y|X].`  
  `X = [b]`  
  `Y = a`  
  `Z = [a, b]`
terms can represent graphs

- \( ?- X = [a|X] \).
  \( X = [a, a, a, a, a, a, a, a, a|...] \)
  Yes
- here \( X \) denotes an infinite or circular list
- this is not allowed in first-order logic; a variable cannot denote a term and one of its subterms; but Prolog omits the “occurs check”

building a knowledge base

- to be used in a computation, facts and rules must be stored in the (dynamic) database
- facts and rules get into the database through assertion and consultation
- consultation loads facts and rules from a file

assertion

- \( ?- \) assert(human(ulyssus)).
- \( ?- \) human(X).
  \( X = ulyssus \)
  Yes
- assertion can be done dynamically
- also retract to remove facts and rules from the DB
- like assignment, change state; avoid when possible

consultation

- \( ?- \) consult(‘family.pl’).
  loads facts and rules from file family.pl
- \( ?- \) [family].
  does the same thing
- \( ?- \) [user].
  lets you enter facts and rules from the keyboard
denotation/meaning of Prolog programs

- A Prolog program defines a set of relations, i.e., specifies which tuples of objects/terms belong to a particular relation.
- In logic, this is called a model.
- Declarative programming is very different from usual procedural programming where programs perform many state changing operations.

denotation of Prolog program e.g.

- fatherOf(john,paul).
- fatherOf(mary,paul).
- motherOf(john,lisa).
- parentOf(X,Y) :- fatherOf(X,Y).
- parentOf(X,Y) :- motherOf(X,Y).
- fatherOf is the relation \{<john,paul>, <mary,paul>\}.
- What is the relation associated with motherOf and parentOf?

rules as procedures

- Rule has form goal :- body.
- Goal or head is like name of procedure.
- Terms on the RHS are like the body of the procedure, the sub-goals that have to be achieved to show that the goal holds.
- The sub-goals will be attempted left-to-right.
- Rule succeeds if all sub-goals succeed.

passing values

- Calling/querying a goal can instantiate its variables.
- A sub-goal's success can bind a variable within it, also binding the same variable in the goal.
- Binding or instantiating a variable is giving it a value.
- Compare to passing values into or out of a procedure.
passing values e.g.

- Assume program:
  
  ```prolog
  motherOf(john,lisa).
  parentOf(X,Y) :- motherOf(X,Y).
  ```

- Queries:
  
  ```prolog
  ?- parentOf(john,X).
  X = lisa  Yes
  ?- parentOf(X,lisa).
  X = john Yes
  ?- parentOf(X,Y).
  X = john, Y = lisa Yes
  ```

- No fixed input and output parameters

relational thinking

- in Prolog, formulate statements about function values as relational facts, e.g.
  
  ```prolog
  factorial(0,1).
  factorial(N,M):- K is N -1, factorial(K,L), M is N * L.
  ```

- to compose functions, e.g. \(Y = f(g(X))\), you must name intermediate results
  
  ```prolog
  fg(X,Y):- g(X,Z), f(Z,Y).
  ```

almost everything is syntactically a term

- lists are terms; what is the functor?

- rules are terms:
  
  ```prolog
  grandfather(X,Y):- father(X,Z), father(Z,Y).
  ```

- What are the functors?

- queries are terms

arithmetic functions

- Prolog retains arithmetic functions as functions (more intuitive):
  
  ```prolog
  ?- X is exp(1). % exp(1) = e^1
  X = 2.71828
  Yes
  ?- X is (4 + 2) * 5.
  X = 30
  Yes
  ```

- How does is compare with =, assignment?
operators

- Some functors are represented by *infix* or *prefix* or *postfix* operators
- Some infix operators: is, =, +, *, /, mod, >, >=, """" :- """", """", etc.
- + and - are both prefix and infix
- :- as prefix is a command, used for declarations
- operators have precedence
- can define our own operators

help is sometimes helpful

?- help(reverse).
reverse(+List1, -List2)
   Reverse the order of the elements in List1 and unify the result with the elements of List2.
+arg: arg is input and should be instantiated.
-arg: arg is output and can be initially uninstantiated; if the query succeeds, the arg is instantiated with the "output" of the query.
?arg: arg can be either input or output

online help

?- help(lists).
No help available for lists
Yes
?- apropos(lists).
merge/3       Merge two sorted lists
append/3      Concatenate lists
Section 11-1  "lists: List Manipulation"
Section 15-2-1 "lists"
Yes
?- help(append/3).
append(?List1, ?List2, ?List3)
   Succeeds when List3 unifies with the concatenation of List1 and List2. The predicate can be used with any instantiation pattern (even three variables).

examples

?- append([a,b],[c],X).
X = [a, b, c]
Yes
?- append(X,[c],[a,b,c]).
X = [a, b]
Yes
?- append([a,b],[c],[a,b,d]).
No
more examples

?- append([a,b],X,Y).
X = _G187
Y = [a, b|_G187]
Yes
?- append(X,Y,Z).
X = []
Y = _G181
Z = _G181 ;
X = [_G262]
Y = _G181
Z = [_G262|_G181] ;
X = [_G262, _G268]
Y = _G181
Z = [_G262, _G268|_G181]

append is an example of a reversible or steadfast predicate (Richard O’Keefe)

reversible programming

- good predicates are steadfast
- they give correct answers even if unusual values are supplied
e. g. variables for inputs, constants for outputs
- non-steadfast predicates require specific arguments to be instantiated (input) or variables (output)

unification

- Prolog matches terms by unifying them, i.e. applying a most general unifier to them
- it instantiates variables as little as possible to make them match, e.g.
  ?- X = f(Y,b,Z), X = f(a,V,W).
  X = f(a, b, _G182)
  Y = a
  Z = _G182
  V = b
  W = _G182

family relations example
family relations

- the database:
  rules
  parent(Parent, Child) :- mother(Parent, Child).
  parent(Parent, Child) :- father(Parent, Child).

  facts
  father('George', 'Elizabeth'). father('George', 'Margaret').
  mother('Mary', 'Elizabeth'). mother('Mary', 'Margaret').

- Note encoding of disjunction

finding all solutions

| ?- parent(Parent, Child).
| Parent = 'Mary',
| Child = 'Elizabeth' ;

| Parent = 'Mary',
| Child = 'Margaret' ;

| Parent = 'George',
| Child = 'Elizabeth' ;

| Parent = 'George',
| Child = 'Margaret' ;

no

how prolog finds solutions

trace] ?- parent(Parent, Child1),
parent(Parent, Child2),
not(Child1 = Child2).
Call: (8) parent(_G313,
_G314) ? creep
Exit: (9) mother('Mary',
'Elizabeth') ? creep
Exit: (8) parent('Mary',
'Elizabeth') ? creep
Redo: (9) mother('Mary',
_G317) ? creep
Exit: (9) mother('Mary',
'Margaret') ? creep
Exit: (8) parent('Mary',
'Margaret') ? creep
Parent = 'Mary'
Child1 = 'Elizabeth'
Call: (8) parent('Mary',
_G317) ? creep
Call: (9) mother('Mary',
_G317) ? creep
Exit: (9) mother('Mary',
'Elizabeth') ? creep
Exit: (8) parent('Mary',
'Elizabeth') ? creep
Redo: (9) mother('Mary',
_G313) ? creep
Exit: (9) mother('Mary',
'Margaret') ? creep
Exit: (8) parent('Mary',
'Margaret') ? creep
Parent = 'Mary'
Child1 = 'Elizabeth'
Child2 = 'Margaret'

Prolog’s query answering process

- a query is a conjunction of terms
- answer to the query is yes if all terms succeed
- A term in a query succeeds if
  ◆ it matches a fact in the database or
  ◆ it matches the head of a rule whose body succeeds
- the substitution used to unify the term and the fact/head is applied to the rest of the query
- works on query terms in left to right order; databases facts/rules that match are tried in top to bottom order
generating permutations

- A permutation \( P \) of a list \( L \) is a list whose first is some element \( E \) of \( L \) and whose rest is a permutation of \( L \) with \( E \) removed.
- \([\,]\) is a permutation of \([\,]\)
- In Prolog:
  \[
  \text{permutation}([\,],[\,]).
  \text{permutation}(L,[E|PR]) :- \text{select}(E,L,R),
  \text{permutation}(R,PR).
  \]

selecting an element from a list

- To select an element from a list, can either select the first leaving the rest, or select some element from the rest and leaving the first plus the unselected elements from the rest.
- In Prolog:
  \[
  \text{select}(X,[X|R],R).
  \text{select}(X,[Y|R],[Y|RS]):- \text{select}(X,R,RS).
  \]

sorting by the definition

- Find a permutation that is ordered
  \[
  \text{sort}(L,P):- \text{permutation}(L,P),
  \text{ordered}(P).
  \text{ordered}([\,]).
  \text{ordered}([E|]).
  \text{ordered}([E1,E2|R]):- E1 \leq E2,
  \text{ordered}([E2|R]).
  \]
- an example of “generate and test"
reverse

- reverse(L, RL) holds if RL is a list with the components of L reversed
- ordinary recursive definition
  reverse([], []).
  reverse([F|R], RL) :- reverse(R, RR),
                        append(RR, [F], RL).
  append([], L, L).
  append([F|R], L, [F|RL]) :-
                        append(R, L, RL).

Tail recursive definition:
reverse(L, RL) :- reverse(L, [], RL).
reverse([], Acc, Acc).
reverse([F|R], Acc, RL) :-
  reverse(R, [F|Acc], RL).

- recursive call is last thing done
- can avoid saving calls on stack

the zebra puzzle

1. There are 5 houses, occupied by politically-incorrect gentlemen of 5 different nationalities, who all have different coloured houses, keep different pets, drink different drinks, and smoke different (now-extinct) brands of cigarettes.
2. The Englishman lives in a red house.
3. The Spaniard keeps a dog.
4. The owner of the green house drinks coffee.
5. The owner of the blue house smokes Pall Mall.
6. The owner of the yellow house drinks tea.
7. The Norwegian lives in a house next to the house where the dog is kept.
8. The green house is just to the right of the ivory house.
9. The owner of the ivory house smokes in a house next to the house where the yellow house is located.
10. The man who smokes Blue Bell lives next to the man who drinks tea.
11. The Norwegian lives in a house next to a house with a fox.

Who owns the zebra and who drinks water?
**Prolog implementation**

- represent the 5 houses by a structure of 5 terms
  
  ```prolog```
  ```
  house(Colour, Nationality, Pet, Drink, Cigarettes)
  ```
  ```
  ```
  ```

- create a partial structure using variables, to be filled by the solution process

- specify constraints to instantiate variables

**house building**

```prolog```
```
makehouses(0,[]).
makehouses(N,[house(Col, Nat, Pet, Drk, Cig)|List]) :- N>0, N1 is N - 1, makehouses(N1,List).
``` or more cleanly with anonymous variables:
```prolog```
```
makehouses(N,[house(_, _, _, _, _)|List]) :- N>0, N1 is N - 1, makehouses(N1,List).
```

Why is this equivalent? (See p. 159.)

**the empty houses**

```prolog```
```
?- makehouses(5, List).
```

**constraints**

- The Englishman lives in a red house.
  ```prolog```
  ```
  house(red, englishman, _, _, _) on List,
  ```

- The Spaniard keeps a dog.
  ```prolog```
  ```
  house(_, spaniard, dog, _, _) on List,
  ```

- The owner of the green house drinks coffee.
  ```prolog```
  ```
  house(green, _, _, coffee, _) on List
  ```

- The ivory house is just to the left of the green house
  ```prolog```
  ```
  sublist2([house(ivory, _, _, _, _), house(green, _, _, _, _)], List),
  ```

- The Chesterfields smoker lives next to a house with a fox.
  ```prolog```
  ```
  nextto(house(_, _, _, fox, _), house(_, _, _, _, _), List),
  ```
defining the on operator

- **on** is a user-defined infix operator that is a version of `member/2`
- `:- op(100,zfy,on).`
  
  `X on List :- member(X,List).`

  amounts to
  
  `X on [X|__].`
  `X on [__|R] :- X on R.`

predicates for defining constraints

- “just to the left of”? “lives next to”?
- define `sublist2(S,L)`
  
  `sublist2([S1, S2], [S1, S2 | __]) .`
  `sublist2(S, [__ | T]) :- sublist2(S, T).`
- define `nextto` predicate
  
  `nextto(H1, H2, L) :- sublist2([H1, H2], L).`
  `nextto(H1, H2, L) :- sublist2([H2, H1], L).`

translating the constraints

- The ivory house is just to the left of the green house
  
  `sublist2( [house(ivory, _, _, _, _),
              house(green, _, _, _, _)], List),`

- The Chesterfields smoker lives next to a house with a fox.
  
  `nextto(house(_, _, _, _, chesterfields),
          house(_, fox, _, _), List),`

looking for the zebra

- Who owns the zebra and who drinks water?
  
  `find(ZebraOwner, WaterDrinker) :-
  makehouses(5, List),
  house(red, englishman, _, _, _) on List,
  ... % all other constraints
  house(_, WaterDrinker, _, water, _) on List,
  house(_, ZebraOwner, zebra, _, _) on List.`

- solution is generated and queried in the same clause
- neither water or zebra are mentioned in the constraints
solving the puzzle

?- [zebra].
% zebra compiled 0.00 sec, 5,360 bytes

Yes
?- find(ZebraOwner, WaterDrinker).

ZebraOwner = japanese
WaterDrinker = norwegian ;

No

how Prolog finds solution

After first 8 constraints:
List = [house(red, englishman, snail, _G251, old_gold),
house(green, spaniard, dog, coffee, _G264),
house(ivory, ukrainian, _G274, tea, _G276),
house(yellow, _G297, _G298, _G299, kools)]

how Prolog solves the puzzle

Then need to satisfy “the owner of the third house drinks milk”, i.e.
List = [_, _, house(_, _, _, milk, _), _, _],
Can’t be done with current instantiation of List. So Prolog will backtrack and find another.

how Prolog solves the puzzle

The unique complete solution is
L = [
house(yellow, norwegian, fox, water, kools),
house(blue, ukrainian, horse, tea, chesterfields),
house(red, englishman, snail, milk, old_gold),
house(ivory, spaniard, dog, orange, lucky_strike),
house(green, japanese, zebra, coffee, parliaments)]
See course web page for code of the example.