CS 314 Principles of Programming Languages

Lecture 23: Logic Programming Paradigm

Prof. Zheng Zhang

Rutgers University

November 30, 2018

Declarative Programming

- **Definition:** A programming paradigm that expresses the logic of a computation without describing its control flow
- Specifies what needs to be done, but not how to do it
- Examples: database query, regular expression, logical language

Algorithm = Logic + Control

Imperative V.S. Logic Programming

- Imperative programming must provide a constructive proof
- Logic programming only needs to specify the logical properties
  - Everything related to control is relegated to the abstract machine
  - The process of ***unification*** forms the basic computation mechanism for logical programming languages

An Example

- Arrange three 1s, three 2s, …, three 9s in a list (which therefore consists of 27 numbers) such that, for each $i \in [1, 9]$, there are exactly i numbers between two successive occurrences of i.
- Therefore, 1,3,1,8,1 could potentially be part of one solution, however, 1,3,1,8,3,1 could not be.

The "how" of the desired solution is not immediately clear!

Implementation in Prolog

- We need a list L. This list will have to contain exactly 27 elements, something that we can specify using a unary predicate:

list_of_27(L) :- L = [_, _, _, _, ….,_]

- The list L will have to contain a sublist in which the number 1 appears followed by any other number, by another occurrence of the number 1, then another number, and finally a last occurrence of the number 1.

[1, X, 1, Y, 1] or [1, _, 1, _, 1]

The Complete Solution

```
• sol(L)  :-
        list_of_27(L),
        sublist([9, _, _, _, _, _, _, _, _, _, 9, _, _, _, _, _, _, _, _, _, 9], L),
        sublist([8, _, _, _, _, _, _, _, _, 8, _, _, _, _, _, _, _, _, 8], L),
        sublist([7, _, _, _, _, _, _, _, 7, _, _, _, _, _, _, _, 7], L),
        sublist([6, _, _, _, _, _, _, 6, _, _, _, _, _, _, 6], L),
        sublist([5, _, _, _, _, _, 5, _, _, _, _, _, 5], L),
        sublist([4, _, _, _, _, 4, _, _, _, _, 4], L),
        sublist([3, _, _, _, 3, _, _, _, 3], L),
        sublist([2, _, _, 2, _, _, 2], L),
        sublist([1, _, 1, _, 1], L),

• list_of_27(L) :-
        L = [_, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _]
```

Potential solutions:

[1, 9, 1, 6, 1, 8, 2, 5, 7, 2, 6, 9, 2, 5, 8, 4, 7, 6, 3, 5, 4, 9, 3, 8, 7, 4, 3]

[1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5, 7]

[1, 8, 1, 9, 1, 5, 2, 6, 7, 2, 8, 5, 2, 9, 6, 4, 7, 5, 3, 8, 4, 6, 3, 9, 7, 4, 3]

[3, 4, 7, 9, 3, 6, 4, 8, 3, 5, 7, 4, 6, 9, 2, 5, 8, 2, 7, 6, 2, 5, 1, 9, 1, 8, 1]

[7, 5, 3, 8, 6, 9, 3, 5, 7, 4, 3, 6, 8, 5, 4, 9, 7, 2, 6, 4, 2, 8, 1, 2, 1, 9, 1]

[3, 4, 7, 8, 3, 9, 4, 5, 3, 6, 7, 4, 8, 5, 2, 9, 6, 2, 7, 5, 2, 8, 1, 6, 1, 9, 1]

Prolog

- **Language constructs**
  - Facts, rules, queries through templates
- **Horn clauses**
  - Goal-oriented semantics
  - Procedural semantics
- **How computation is performed?**

Prolog

- **As a database**
  - Start with program as a database of facts
  - Simple queries with constants and variables ("binding"), conjunctions and disjunctions
  - Add program rules to derive additional facts
  - Two interpretations
    - Declarative: based on logic
    - Procedural: searching for answers to queries
      Search trees and rule firings can be traced

Facts and Queries

Facts:

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

Queries:

?-likes(al,eve).
yes

?-likes(al, pie)
no

?-likes(eve,al).
no

?-likes(person,food).
no

?-likes(al,Who).
Who=eve

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

?-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

Facts and Queries

Facts:

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

More Queries:

?-likes(eve,W), likes(W,V).
W=eve,V=pie ; W=eve,V=tom ; W=eve,V=eve

?-likes(eve,W), person(W), food(V).
W=tom,V=pie ; W=tom,V=apple

?-likes(eve,V), (person(V); food(V)).
V=pie ; V=tom ; no

?-likes(eve,W), \+likes(al,W).
W=pie ; W=tom ; no

Solving the Problem

- Facts alone do not make interesting programs possible. Need variables and deductive rules.

?-female(X).	a query
X = alice ;	; asks for more answers
X = victoria ;	if user types <return> then no
no	more answers given

- Variable X has been *unified* to all possible values that make female(X) true.
- Variables capitalized, predicates and constants are lower case

```
male(albert).
female(alice).
male(edward).
female(victoria).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
```

Back to the Example

```
male(albert).
female(alice).
male(edward).
female(victoria).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
```

```
sister_of(X,Y)    :-    female(X),
                        parents(X,M,F),
                        parents(Y,M,F).
```

a rule

```
?- sister_of(alice,Y).
Y = edward
```

```
?- sister_of(alice, victoria).
false
```

First-order Predicate Calculus

- **Constants**
  - Represent entities (things, not functions or relations)
  - In prolog: start with lower case
    - albert, my_house
- **Variables**
  - Stand for constants
  - In prolog: start with upper case or _
    - X, House, _xyz, _321

First-order Predicate Calculus

- **Functors**
  - Represent a function from entities to an entity.
    - "Function" = "mapping" as in math, not a computation
  - In Prolog: start with lower case like constants.
    - In fact, a constant is just a functor with no arguments
- **Terms**
  - Represent an entity
  - Constant, Variable, or <functor>[(<term> {, <term>})]
    - father(albert) might represent the father of albert
    - successor(victoria) might represent the successor of victoria
    - sum(1, 2) might represent the sum of 1 and 2

First-ordre Predicate Calculus

- **Predicates**
  - Represent a function from entities (terms) to a boolean
  - In Prolog: start with lower case like functors.
- **Atoms**
  - Logical statement without and, or, not, etc.
    <predicate>(<term> {, <term>})
    - older(father(Person), Person)
    - square(2, 4)

Horn Clauses

- **A Horn Clause is: $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$**
  - Antecedents (h’s): conjunction of zero or more conditions which are atoms
  - Consequent (c): an atomic formula
- **Meaning of a Horn clause:**
  - The consequent is true if the antecedents are all true
  - c is true if h_1, h_2, h_3, . . . , and h_n are all true
  - likes(al, F) ← likes(eve, F), food(F).

Horn Clauses

- In Prolog, a Horn clause $c \leftarrow h_1 \wedge \ldots \wedge h_n$
  is written c :- h_1 , ... , h_n.
- Horn Clause is a Clause
- Consequent is a Goal or a Head
- Antecedents are Subgoals or Tail
- Horn Clause with No Tail is a Fact
  male(edward). *dependent on no other conditions*
- Horn Clause with Tail is a Rule
  father(albert, edward) :-
  male(edward), parents(edward, M, albert).

Variables in Prolog Clauses

- The consequent and antecedents of a clause may have variables:
  $c(X_1, \ldots X_i)$:- $h(X_1, \ldots X_i, Y_1, \ldots, Y_j)$

Some variables (Y's) in tail but not in head, called "*auxiliary*" variables

Declarative Semantics

- Interpret facts & rules as logic statements
- Example rule
  sister_of(X,Y) :- female(X), parents(X,M,F), parents(Y,M,F).
  corresponds to logical properties:

  X is the sister of Y, if X is female, and
  there are M and F who are X's parents, and
  Y's parents

Note that auxiliary variables are existentially quantified

- A query is a conjunction of atoms, to be proven
  - If query has no variables and is provable, answer is true
  - If query has variables, proof process causes some variables to be bound to values (called a ***substitution***); these are reported

Examples

```
sister_of(X,Y):-
        female(X), parents(X,M,F), parents(Y,M,F).

?-sister_of(alice,Y).
Y = edward

?-sister_of(X,Y).
X = alice
Y = edward ;
X = alice
Y = alice ; /* what went wrong here? */
no
```

```
(1) male(albert).
(2) female(alice).
(3) male(edward).
(4) female(victoria).
(5) parents(edward, victoria, albert).
(6) parents(alice, victoria, albert).
```

Procedure Semantics

- **Interprets facts & rules as an algorithm to do something**
  sister_of(X,Y):- female(X), parents(X,M,F), parents(Y,M,F).
  ?- sister_of(Sis, Sib)

- **Find Sis & Sib such that sister_of(Sis, Sib) is proven**
  - Bind Sis to X, Sib to Y
  - First find an X to make female(X) true
  - Second find an M and F to make parents(X,M,F) true for that X.
  - Third find a Y to make parents(Y, M, F) true for those M,F

Prolog Rule Ordering and Unification

- Rule ordering (from first to last) used in search
- Clauses solved in left-to-right order
- Unification requires all instances of the same variable in a rule to get the same value
- Unification does not require differently named variables to get different values: sister_of(alice, alice)

Fixed Example

```
sister_of(X,Y):-
        female(X), parents(X,M,F), parents(Y,M,F), \+(X==Y).

?-sister_of(alice,Y).
Y = edward

?-sister_of(X,Y).
X = alice
Y = edward ;
no
```

=   means unifies with
== means same in value

Negation as Failure

- **\+(P) succeeds when P fails**
  - Called *negation by failure*.
  - Read as “P unprovable”, not “P false”

Backtracking in Prolog

eat(wolf, lamb)

eat(lamb, grass).

plant(grass).

eat(lion,X) :- eat(X, Food), plant(Food).

eat(lion, X)

eat(X, Food) plant(Food)

X = wolf, Food = lamb **Food = lamb**

eat(wolf, lamb) ?? fail

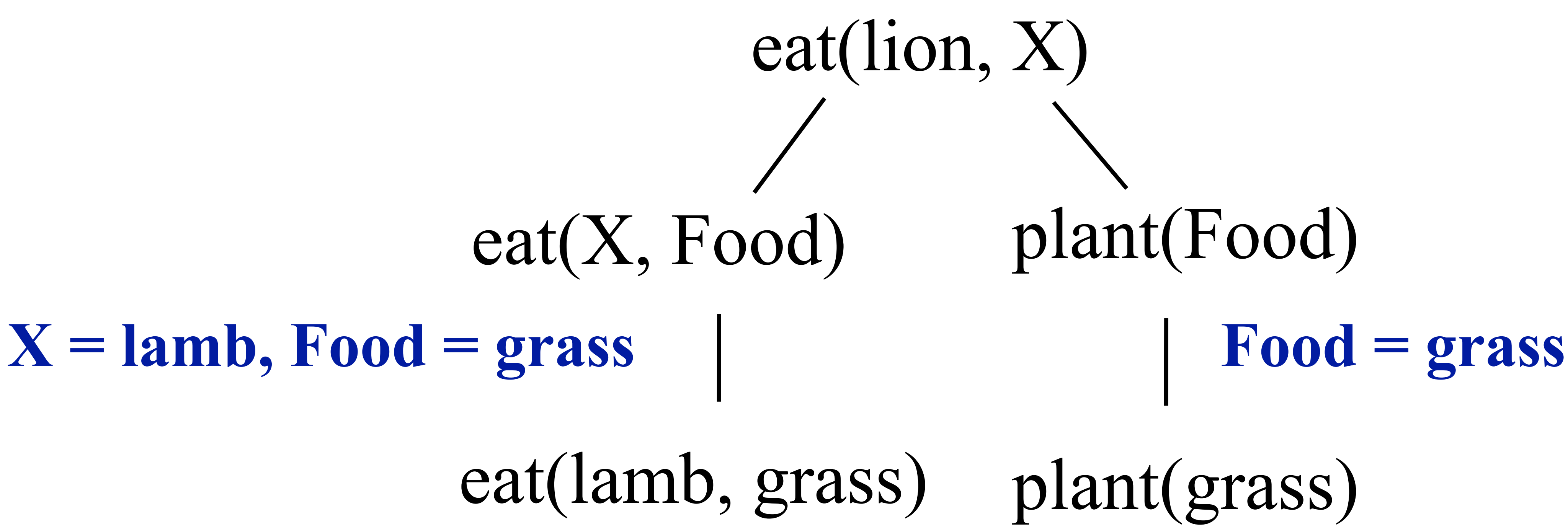
Backtracking in Prolog

eat(wolf, lamb)

eat(lamb, grass).

plant(grass).

eat(lion,X) :- eat(X, Food), plant(Food).



Cuts

- The goal ! (read “cut”) always succeeds, but throws away some choice points
- foo:-bar, !, baz when you see the !, throw away all choice points since began goal foo

Penalty of Efficiency

- The computation performed by the language's abstract machine is very complex.
- The interpreter must try the various substitution options until it finds the one that satisfies all the conditions.
- In these search processes, a ***backtracking*** mechanism is used. When the computation arrives at a point which it cannot proceed, the computation that has been performed is undone such that a decision can be reached, if it exists, at which an alternative is chosen (if the alternative does not exist, the computation terminates in failure).
- In general, this search can have exponential complexity.

Reading

- **Scott, Chapter 12.1 - 12.3**