Principles of Programming Languages

Topic: Logic Programming

Professor Lou Steinberg
Logic Programming

• True facts:
  – If I was born in year B, then in year Y on my birthday I turned Y-B years old
  – I turned 54 on my birthday in 2005.

• From these facts you can answer the question
  – What year was I born?

• The same facts & query in a programming language:
  
  age_in_birthday_year = birth_year - birthday_year;
  age_in_birthday_year  = 54;
  birthday_year = 2005;
  print(birth_year); /* ??? */
Logic Programming

• Goal of logic programming:
  – State true facts
  – Have computer figure out answer to a query

• Reality:
  – State true facts
    In slightly-restricted First Order Predicate Calculus
  – Have the computer figure out answer to a query
    If you wait long enough
Example: Sorting

Lout is a sorted version of Lin if
• Lout is a permutation of Lin, and
• Lout is in order

Resulting algorithm:
• For each permutation Lp of Lin:
  If Lp is in order, then return Lp as Lout
• This is $O(N! \times N)$
  – to sort 20 numbers would take $48,658,040,163,532,800,000$ comparisons
    $< !!!!! >$
Example: Sorting

Lout is a sorted version of Lin if
• < a logical description of merge sort >
  See merge.pl

Resulting algorithm:
• merge sort: \( O(n \log n) \)
Logic Programming

• Goal of logic programming:
  – State true facts
  – Have computer figure out answer to a query

• Problem: this is hard

• Alternate goal:
  – Separate specification of facts from specification of how to use them
Prolog

• Language constructs
  – Facts, rules, queries through templates

• Horn clauses
  – Goal-oriented semantics
  – Procedural semantics

• How computation is performed?
• Comparison to logic programming
Prolog

• As a database
  – Start with program as a database of facts
  – Simple queries with constants and variables ("binding"), conjunctions and disjunctions
  – Add to 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

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

predicates

constants
Queries (Asking Questions)

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

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

query

?-likes(al, eve).

who = eve

?-likes(eve, W).

w = pie ;

w = tom ;

w = eve ;

no

force search for more answers

variable

answer with variable binding
Harder Queries

| 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
Harder Queries

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

?-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 or

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

?-likes(eve, W), \+likes(al, W).
W=pie ; W=tom ; no
• What if you want to ask the same question often?

**Add a rule to the database:**

```
rule1:-likes(eve,V),person(V).
```

```
?-rule1.
yes
```
rules

likes(eve, pie).    food(pie).
likes(al, eve).    food(apple).
likes(eve, tom).    person(tom).
likes(eve, eve). ___________________________
rule1:-likes(eve,V),person(V).
rule2(V):-likes(eve,V),person(V).

?-rule2(H).
H=tom ; no
?-rule2(pie).
no
Note rule1 and rule2 are just like any other predicate!

CS 314, LS, BR, AB: Prolog
Queen Victoria Example

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

?- [family].  loads file
yes

?- male(albert).  a query
yes

?- male(alice).  
no

?- parents(edward,victoria,albert).  
yes

?- parents(bullwinkle,victoria,albert).  
no
**Queen Victoria Example, cont.**

- Problem: facts alone do not make interesting programs possible. Need variables and deductive rules.

```prolog
?- female(X). ; asks for more answers
X = alice ;
X = victoria ; if user types <return> then no more answers given
false when no more answers left, return false
```

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

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-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 \(<\text{functor}>[(<\text{term}>\{, <\text{term}>\})\)]
    • 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-order 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(X, 4)
Horn Clauses

• A Horn Clause is: \( c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \)
  
  – 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 \( h_1, h_2, h_3, \ldots, \) and \( h_n \) are all true

\[
\text{likes(al, F)} \leftarrow \text{likes(eve, F), food(F).}
\]
Horn Clauses

- In Prolog, a Horn clause \( c \leftarrow h_1 \wedge \ldots \wedge h_n \) is written \( c :\neg h_1, \ldots, h_n \).
- 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)
  :\neg male(edward), parents(edward,M,albert).
Variables in Prolog Clauses

• The consequent and antecedents of a clause may have variables:

\[ c(X_1, \ldots, X_i) \leftarrow h(X_1, \ldots, X_i, Y_1, \ldots, Y_j) \]

Some variables (Y’s) in tail but not in head: called “auxiliary” variables
Declarative Semantics

- Interpret facts & rules as logic statements
- EG rule

\[ \text{sister_of}(X,Y) :\sim \\
\text{female}(X), \text{parents}(X,M,F), \text{parents}(Y,M,F). \]

corresponds to logical formula:

\[ \forall X,Y . \text{sister_of}(X,Y) \leftarrow \\
\exists M,F . \text{female}(X), \text{parents}(X,M,F), \text{parents}(Y,M,F). \]

/* 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
Declarative Semantics

• 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
Example

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).
Procedural 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
Procedural Semantics

• This algorithm is recursive; each find works just like this one
  – But on a new “copy” of the facts+rules, starting from top.
  – Every time rule is used, rename variables
  – Eventually, each find must be resolved by appealing to facts.

• Process is called backward chaining.
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)
Example

\[\text{sis}(X, Y) : - \text{female}(X), \text{parents}(X, M, F), \]
\[\quad \text{parents}(Y, M, F), \plus{(X==Y)}. \]

?-sis(X, Y).  \textit{last subgoal disallows }X,Y\textit{ to have same value}

\textit{X=alice}
\textit{Y=edward}  ;
\textit{no}

= means \textit{unifies with}

== means \textit{same in value}
Negation as Failure

• \(\neg(P)\) succeeds when \(P\) fails
  – Called negation by failure.
  – Read as “\(P\) unprovable”, not “\(P\) false”
Transitive Relations

parents(jane,sally,bob). parents(john,sally,bob).
parents(sally,mary,al). parents(bob,ann,mike).
parents(mary,lee,joe).

Y is ancestor of X

ancestor(X,Y):- parents(X,Y,_).
ancestor(X,Y):- parents(X,_,Y).
ancestor(X,Y):- parents(X,Z,_),ancestor(Z,Y).
ancestor(X,Y):- parents(X,_,Z),ancestor(Z,Y).

?-ancestor(jane,X).
X= sally ;  X=ann ;
X= bob ;  X=mike ;
X= mary ;
X= al ;
X= lee ;
X= joe ;
Transitive Relations

• What if you wrote

\texttt{ancestor(X,Y):-ancestor(Z,Y),parents(X,Z,_).}

See \texttt{tc.pl}
Logic vs Prolog

• Logic: **Nondeterministic**
  - Arbitrarily choose rule to expand first
  - Arbitrarily choose subgoal to explore first
  - Results don't depend on rule and subgoal ordering

• Prolog: **Deterministic**
  - Expand first rule first
  - Explore first (leftmost) subgoal first
  - Results may depend on rule and subgoal ordering
Backtracking

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

\[ \begin{align*}
  \text{eat}(\text{lion}, \text{Y}) \\
  \text{Y} = \text{X} \\
  \text{eat}(\text{X}, \text{Food}) & \quad \text{plant}(\text{Food})
\end{align*} \]
Backtracking

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

Y = X = lamb

X = lamb, Food = grass

Food = grass

Food = grass

Backtracking

\text{plant(grass)}.
\text{eat(lamb,grass)}.
\text{eat(lion,X)} :-
\hspace{1cm} \text{eat(X,Food),plant(Food)}.

\begin{center}
\begin{tikzpicture}
  \node (eatlion) {eat(lion,Y)};
  \node (xfood) [below left of=eatlion, yshift=-1cm] {eat(X, Food)};
  \node (plantfood) [right of=xfood, xshift=2cm] {plant(Food)};
  \node (xfoodgrass) [below of=xfood, yshift=-1cm] {X = lamb, Food = grass};
  \node (plantgrass) [right of=xfoodgrass, xshift=2cm] {plant(grass)};
  \node (xfoodgrassfood) [below of=xfoodgrass, yshift=-1cm] {Food = grass};
  \node (xfoodgrassfoodgrass) [below of=xfoodgrassfood, yshift=-1cm] {Food = grass};

  \draw (eatlion) -- (xfood);
  \draw (xfood) -- (xfoodgrass);
  \draw (xfoodgrass) -- (xfoodgrassfood);
  \draw (xfoodgrassfood) -- (xfoodgrassfoodgrass);
  \draw (xfoodgrassfoodgrass) -- (plantfood);
  \draw (plantfood) -- (plantgrass);
\end{tikzpicture}
\end{center}

Y = X = lamb

X = lamb, Food = grass

Food = grass

Food = grass

plant(grass)
Backtracking

plant(grass).
eat(wolf, lamb)
eat(lamb, grass).
eat(lion, X):-
  eat(X, Food), plant(Food).

\[
\begin{align*}
& \text{eat(lion, Y)} \\
& \text{Y} = \text{X} \\
& \text{eat(X, Food)} \quad \text{plant(Food)}
\end{align*}
\]
Backtracking

eat(wolf, lamb)
eat(lamb, grass).
plant(grass).
eat(lion, X):-
    eat(X, Food), plant(Food).

\[ \text{eat(lion, } Y) \]
\[ \text{Y = X = wolf} \]
\[ \text{eat(X, Food)} \quad \text{plant(Food)} \]
\[ \text{X = wolf, Food = lamb} \]
\[ \text{eat(wolf, lamb)}^{*} \]
Backtracking

eat(wolf, lamb)
eat(lamb, grass).
plant(grass).
eat(lion, X) :-
  eat(X, Food), plant(Food).

\[
\begin{align*}
  Y &= X = \text{wolf} \\
  X &= \text{wolf}, \ Food = \text{lamb} \\
  \text{eat(wolf, lamb)} &\ast \quad ?? \text{ fail}
\end{align*}
\]
Backtracking

```
eat(wolf, lamb)
eat(lamb, grass).
plant(grass).
eat(lion, X): -
        eat(X, Food), plant(Food).
```

```
Y = X = lamb
X = lamb, Food = grass
Food = grass
```

```
eat(lion, Y)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```

```
eat(lion, Y)
eat(X, Food) plant(Food)
```
Backtracking

• See guardpet.pl
Cut

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