

Administrative Issues

CSC324 Principles of Programming Languages (Week 8)

Yilan Gu

yilan@cdf.toronto.edu

<http://www.cs.toronto.edu/~yilan/324f09>

- Assignment 3 will due on Nov. 2, 2009 6:00pm sharp
- Nov. 5, 6:00-7:00pm, Term test 2
 - Cheating sheet
 - 20% policy
 - Pen for remark
 - Contents from last test till today
 - Check the web for details
 - Assignment 3 solution will be posted on Nov. 4 once submission is closed for late assignments

Logic Programming and Prolog

- Logic programming languages are not procedural or functional.
 - Specify relations between objects
 - larger(3,2).
 - father(tom,jane).
 - Separate logic from control:
 - Programmer declares what facts and relations are true.
 - System determines how to use facts to solve problems.
 - System instantiates variables in order to make relations true.
 - Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
 - Higher-level than imperative languages

Logic Programming

Jump Right In: Prolog

Suppose we state these facts:

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

We can then make queries:

?- male(albert).

Yes

?- male(victoria).

No

?- female(Person).

Person = alice;

Person = victoria;

No

?- parent(Person, edward).

Person = albert;

Person = victoria;

No

Functional Programming v.s. Logic Programming

- In FP, we program with functions.
 $f(x, y) = x+y$
Given a function, we can only ask one kind of question:
 - Here are the argument values;
tell me what is the function's value.
- In LP, we program with relations.
 $\text{sum}(x, y, z) = \{ (1, 2, 3), (9, 5, 14), (0, 0, 0) \}$
Given a predicate, we can ask many kinds of questions:
 - Here are some of the argument values;
tell me what the others have to be in order to make a true statement
 - Constants are special function terms with no argument.
 - Functions are used as terms, representing objects:
office(X): office of X

Jump Right In: Prolog

We can also state rules, such as this one:

sibling(X, Y) :- parent(P, X),
parent(P, Y).

Then the queries become more interesting:

?- sibling(albert, victoria).

No

?- sibling(edward, Sib).

Sib = edward;

Sib = alice;

Sib = edward;

Sib = alice;

No

LP: resolution in predicate logic

- We would like to infer new propositions (e.g. facts) from some existing set of propositions.

- An inference rule that can be applied atomically is called a **resolution**

– E.g.

Given: $P1 \leftarrow P2$, $Q1 \leftarrow Q2$
 $P1 \equiv Q2$

Alternatively: $T \leftarrow P2$, $Q1 \leftarrow T$

New fact: $Q1 \leftarrow P2$

- Resolution gets more complex if variables/values are involved:

– To use resolution with variables, we will need to find values for variables that allow matching to proceed.

– E.g.

Given: $F(X,Y) \leftarrow P2(Y,X)$

$Q1(\text{foo}) \leftarrow F(\text{foo}, \text{bar})$

New fact: $Q1(\text{foo}) \leftarrow P2(\text{bar}, \text{foo})$

LP: horn clause

- Logic programming is heavily based on horn clauses:

$$c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \dots \wedge h_n$$

- Antecedents (h's): conjunction of zero or more conditions which are **atomic** constructs in predicate logic (also called first-order logic).
 - Consequent(c): an atomic construct in predicate logic
 - Also represented as $h_1 \wedge h_2 \wedge h_3 \wedge \dots \wedge h_n \supset c$ in logic
- 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, \dots are all true

LP: specifying non-horn rules

- Examples:

$$\begin{aligned} \neg A \leftarrow \neg B &\equiv \neg A \vee \neg(\neg B) \\ &\equiv \neg A \vee B \\ &\equiv B \vee \neg A \\ &\equiv \underline{B \leftarrow A} \quad (\text{horn-clause}) \end{aligned}$$

$$\begin{aligned} A \leftarrow (B \vee C) &\equiv A \vee \neg(B \vee C) \\ &\equiv A \vee (\neg B \wedge \neg C) \\ &\equiv (A \vee \neg B) \wedge (A \vee \neg C) \\ &\equiv \underline{(A \leftarrow B) \wedge (A \leftarrow C)} \quad (\text{horn-clause}) \end{aligned}$$

LP: specifying non-horn rules

- Many non-horn rules can be transformed to horn form using one of two methods:

- logical equivalence
- Skolemization

- Logical equivalence:**

- Uses the following logical laws:

- Negation** $\neg\neg A \equiv A$
- De Morgan's Law** $\neg(A \vee B) \equiv \neg A \wedge \neg B$
 $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- Distributive Property** $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
 $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
- Absorption Law** $A \vee (A \wedge B) \equiv A$
 $A \wedge (A \vee B) \equiv A$
- Implication Laws** $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$
 $A \leftarrow B \equiv A \vee \neg B$

LP: specifying non-horn rules – cont'd

- Examples:

$$\begin{aligned} A \leftarrow (B \leftarrow C) &\equiv A \vee \neg(B \leftarrow C) \\ &\equiv A \vee \neg(B \vee \neg C) \\ &\equiv A \vee (\neg B \wedge \neg \neg C) \\ &\equiv A \vee (\neg B \wedge C) \\ &\equiv (A \vee \neg B) \wedge (A \vee C) \\ &\equiv (A \leftarrow B) \wedge (A \vee C) \quad (\text{non-horn}) \end{aligned}$$

- In general, rules of the form

$$(\forall X)[(A_1 \vee \dots \vee A_n) \leftarrow (B_1 \wedge \dots \wedge B_m)]$$

cannot be converted to horn-clause

LP: specifying non-horn rules – cont'd

- **Horn clause** $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \dots \wedge h_n$
 - What are we going to do about quantifiers?
- **Skolemization:**
 - Variables bound by existential ($\exists X$) quantifiers which are not inside the scope of universal quantifiers can simply be replaced by constants:
 - $(\exists X) [X < 3]$ becomes $c < 3$
 - $(\exists X) \text{mother}(\text{john}, X)$ becomes $\text{mother}(\text{john}, m)$
 - When the existential quantifier ($\exists X$) is inside a universal quantifier ($\forall Y$), the bound variable must be replaced by a *function* of the variables bound by universal quantifier ($\forall Y$).
 - $(\forall X) [X=0 \vee \exists(Y) [X=Y+1]]$ becomes $(\forall X) [X=0 \vee X = f(X) + 1]$
 - $(\forall X) [\text{person}(X) \rightarrow (\exists Y) \text{mother}(X, Y)]$
becomes $(\forall X) [\text{person}(X) \rightarrow \text{mother}(X, m(X))]$

LP: horn clause made easy!

- **One post condition can have multiple preconditions (conjunctions of atomic predicates), all postconditions considered as disjunctive relation:**

$p(X) \leftarrow h_1(X, Y) \wedge \dots \wedge h_n(X).$
 $p(X) \leftarrow \dots$
 $p(X) \leftarrow \dots$
 $p(X) \leftarrow \dots$
 $p(X) \leftarrow \dots$

- **We can assume the following when writing horn-clauses:**
 - p is the program name
 - h_1, \dots, h_n, \dots are the subprogram names
 - X is a parameter of the program
 - Y is a local variable

LP: specifying non-horn rules – cont'd

- **Skolemization:**
 - Non horn formulas like $(\exists X) A(X)$ can be converted to horn-clause by introducing a *skolem constant* and/or *skolem function*. The resulting clause is *almost* the same thing.
- **Why does skolemization work?**
 - We only need $\exists X$ because we don't have a name for X . By creating artificial names (*skolem names*), we can eliminate many \exists 's and convert many formulas to horn clause.

Horn Clause Summary

- Most but not all first-order logic formulas can be represented using horn clauses
- Each horn clause is of the form
 $\langle \text{atom} \rangle \leftarrow \langle \text{atom}_1 \rangle \wedge \dots \wedge \langle \text{atom}_n \rangle$
- One post condition can have multiple preconditions (conjunctions of atomic predicates), all postconditions considered as disjunctive relation
- Prolog rules are based Horn clause structures (but with some extension)

Prolog I

Prolog: data types – quick intro

- **Simple**

- Constants :

Numbers: integer, floating point,...

Atoms: alphabetic sequence starting with a lower case letter (e.g. apple)

- Variables:

Variables start with capital letters or underscore

- **Complex**

- Structures
- Lists (later, prob. next lecture)

Prolog:- Programmation en logique

- **The first and most popular logic programming language**

- Invented by Alain Colmerauer and Phillipe Roussel at the University of Aix-Marseille in 1971 (France)

- **Characteristics:**

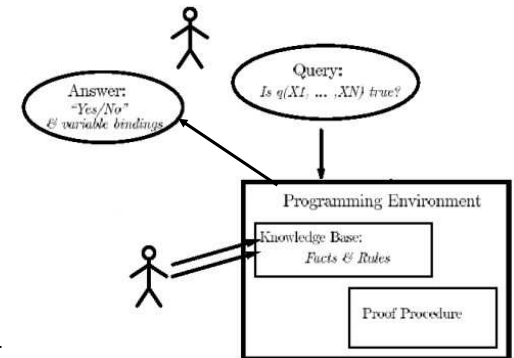
- Is **very weakly typed**
- Has **no data abstraction**
- Has **no functional abstraction!**
- Has **no mutable state**
- Has **no explicit control flow**

- **So, how do you program?**

- Load facts/rules into interpreter
- Make queries to see if a fact is:
 - in the knowledge-base or
 - can be implied from existing facts

or rules

- **Prolog is really an engine to *prove theorems***



Prolog: based on horn clause structure

- **Recall**

$$c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \dots \wedge h_n$$

- **Syntax:** <head> :- <body>.

- You can conclude that <head> is true, if you can prove that <body> is true
- The symbol :- is read as **if**

- **3 types of clauses:**

- Facts
- Rules
- Queries

Prolog: facts

- A fact is a clause with an empty body
- Syntax
 <head>.
- What makes a fact a fact?
- Examples
 - Exams exams.
 - Assignments assignments.
 - Taxes taxes.
 - The earth is round. round(earth).
 - The sky is blue. blue(sky).
 - The sun is hot. hot(sun).
 - Mary is a female. female(mary).
 - Beethoven lived between 1770 & 1827. person(beethoven,1770,1827).

Prolog: rules

- A rule in Prolog is in a full horn clause format:
 $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \dots \wedge h_n$
- Syntax:
 $\underbrace{rel_1}_{\text{head}} \text{ :- } \underbrace{rel_2, rel_3, \dots, rel_n}_{\text{body}}.$
 - If I know that all those relations (those in the body) hold, then I also know that this LHS relation (in the head) holds.
- Examples:
 - If there is smoke there is fire
 fire :- smoke.
 - If the course is boring, I leave
 leave(i) :- boring(course).
 - Joe is going to kill the teacher if he fails CSC324.
 kills(joe, X) :- fails(joe,csc324), teaches(X,csc324).

Prolog: facts – cont'd

- Facts about facts:
 - Full stop “.” at the end of every fact.
 - The number of arguments in a fact is called **arity**.
 - E.g. female(mary). is an instance of female/1 (functor female, arity 1)
 - Facts with different number of arguments are distinct
 - E.g. female(mary,may). is different from female(mary).

Prolog: rules – cont'd

- Examples:
 - X is female if X is the mother of anyone.
 female(X) :- mother(X,_). % avoid singleton variables by using _.
 - X is the sister of Y, if X is female and X's parents are M and F, and Y's parents are M and F
 sister_of(X,Y):- female(X),parents(X,M,F),parents(Y,M,F).
 % in general, how we interpret the rule in first-order logic (predicate logic)?
- When to use rules?
 - Use rules to say that a particular fact depends on a group of facts.
 - Use rules to deduce new facts from existing ones.
- Rules of rules:
 - The head of the rule consist of at most one predicate
 - The body of the rule is a finite sequence of literals separated by ‘!’ (which means conjunction *and*)
 - Rules always end with a period “.”

Prolog: queries

- A query is a clause with an empty head.

$$\leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \dots \wedge h_n$$

- **Syntax**

?- <body>.

- Try to prove that <body> is true
- The goal is represented to the interpreter as a question.

- **Examples**

?- round(earth). % is it true that the earth is round?
 % (or simpler than that: is the earth round?)

?-round(X). % is it true that there are entities which are round?
 % (or simpler than that: what entities are round?)

Prolog: simple types - constants

- There are two types of constants: *atoms* and *numbers*.
- **Atoms:**
 - Alphanumeric atoms: *alphanumeric sequence starting with a lower case letter*
 - E.g.: apple a1 apple_cart
 - Special atoms
 - E.g. ! ; []
 - Symbolic atoms: *sequence of symbolic characters*
 - E.g. & < > * - +
 - Quoted atoms: *sequence of characters surrounded by single quotes*
 - Can make anything an atom by enclosing it in single quotes.
 - E.g. 'apple' 'hello world'
- **Numbers:**
 - Integers and Floating Point numbers
 - E.g. 0 1 9821 -10 1.3 -1.3E102

Prolog: queries – cont'd

- **Examples**

?- composer(beethoven,1770,1827).

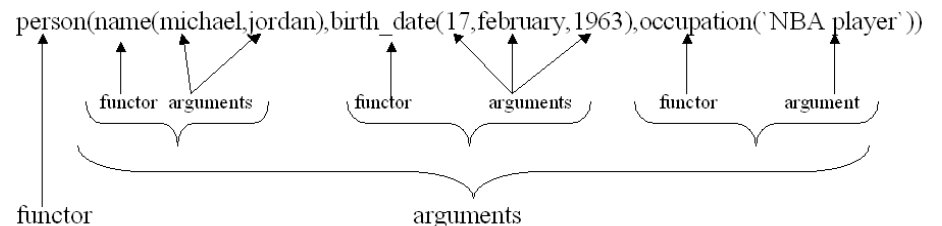
- is it true that beethoven was a composer
 who lived between 1770 and 1827

?- owns(john,book). - is it true that john owns a book?
 (simpler: does john own a book?)

?-owns(john,X). - is it true that john owns something?
 (simpler: does john own something?)

Prolog: complex types - structures

- **Recall: what's a functional term?**
functor(some-parameters) e.g. office(mary)
- **We can construct complex data structures using nested *functional terms*.**
 - Represents a statement about the world
- **Example:**
 - A person has; name: first name, last name - birth date: day, month, year & occupation



Prolog: complex types - structures

Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query:

"Retrieve everything that John owns."

i.e., Find X such that owns(john,X) is true.

```
answers: X = car(red,corvette)
         X = cat(black,siamese,sylvester)
```

Query:

"Retrieve the colour and make of John's car."

i.e., owns(john,car(Colour,Make))

```
answer: Colour = red
        Make = corvette
```

Prolog: complex types - structures

Same Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query: "Who owns a copyright?"

i.e., Find values for Who so that
 $\exists X, Y$ owns(Who,copyright(X,Y)) is true.

```
answers: Who = elvis
        Who = tolstoy
```

Prolog: complex types - structures

Same Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query: "Who owns a red car?"

i.e., Find values for Who so that
 \exists Make owns(Who,car(red,Make)) is true.

```
answers: Who = john
        Who = elvis
```

Prolog: an example

Facts

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

variable

query
 ?-likes(al, pie).

```
no
?-likes(al, eve).
yes
?-likes(eve, al).
no
?-likes(person, food).
no
```

answer

?-likes(al, Who).

Who=eve

?-likes(eve, W).

W=pie ;

W=tom ;

W=eve ;

no

answer with
 variable binding

force search for
 more answers

Prolog: example – cont'd

Facts

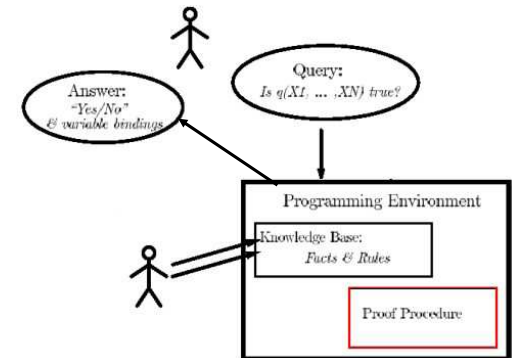
```
likes(eve, pie).    food(pie).
likes(al, eve).    food(apple).
likes(eve, tom).   person(tom).
likes(eve, eve).
```

```
?-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
```

and

Prolog: proof procedure

- Two main processes:
 - Unification
 - Top-down reasoning



Prolog: unification

- First step in proof procedure
- Prolog tries to satisfy a query by *unifying* it with some conclusion and see if it is true!
- Process of finding these suitable "assignments" of values to variables is called *unification*
 - It is really a process of pattern matching to make statements identical
 - Somewhat similar to variable bindings in imperative world and to pattern matching in Scheme.

Prolog: unification – cont'd

- Rules of unification:

Object 1	Object 2	example		result
constant	free var.	4	X	X=4
bound variable	free variable	X	Y	Y gets the value of X
free variable	bound variable	X	Y	X gets the value of Y
bound variable	constant	X	"b"	fails if X has a value different then "b"
compound object	compound object	f(X,Y)	f(2,3)	X=2, Y=3
compound object	compound object	f(q(2,X),3)	f(P,3)	succeeds if P is free, and P=q(2,X) . (.. more possibilities)
compound object	compound object	f(3,X)	q(3,X)	fails, due to different functors (p is not q)

Prolog: unification – cont'd

- **Rules of unification:**
 - A constant unifies only with itself, it cannot unify with any other constant.
 - Two structures unify iff they have the same name, number of arguments and all the arguments unify.
 - Unification requires all instances of the same variable in a rule to get the same value

Prolog: unification – cont'd

- **Examples:**

```

a(b, C, d, E)
with x( ... )  doesn't unify: a and x differ

a(b, C, d, E)
a( -, -, -)    no: different # of args

a(b, C, d, E)
a(j, f, G, H)  no: b ≠ j

a(b, C, d, E)
a(b, f, G, H)  yes: by either {C ↦ f, G ↦ d, H ↦ E}
               or {C ↦ f, G ↦ d, E ↦ H}

a(pred(X, j))
a(pred(k, j))  yes: {X ↦ k}

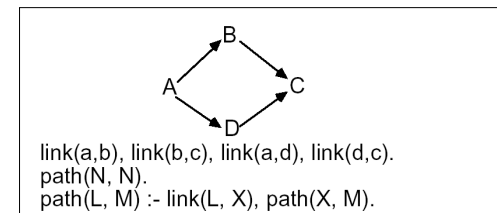
a(pred(X, j))
a(B)           yes: {B ↦ pred(X, j)}
    
```

Prolog: unification – cont'd

- **Examples:**
 - Does $p(X,X)$ unify with $p(b,b)$?
 - Does $p(X,X)$ unify with $p(b,c)$?
 - Does $p(X,b)$ unify with $p(Y,Y)$?
 - Does $p(X,Z,Z)$ unify with $p(Y,Y,b)$?
 - Does $p(X,b,X)$ unify with $p(Y,Y,c)$?
 - To make the third arguments equal, we must replace X by c
 - To make the second argument equal, we must replace Y by b.
 - So, $p(X,b,X)$ becomes $p(c,b,c)$, and $p(Y,Y,c)$ becomes $p(b,b,c)$.
 - However, $p(c,b,c)$ and $p(b,b,c)$ are not syntactically identical.

Prolog: example 2

- **Facts & rules:**



- **Posing queries:**

Based on our logical encoding of the graph, we can then write queries:

```

?- path(a,c)
yes

?- path(c,a)
no

?- path(a,X), path(X,c)
X = a
X = b
X = c
X = d
    
```

Notice that we didn't write a graph traversal algorithm, and we didn't hard code the set of questions we can ask in advance. We just define what a graph is...