CS 321
Programming Languages and Compilers

Prolog
• PROgramming LOGic
• Algorithm = Logic + Control
• Logic programming deals with computing relations rather than functions.
• To understand Prolog one must understand
  – logic: the rules of deduction in first order logic
  – control: the way Prolog implements this logic, i.e. the search strategy that Prolog adopts.
What is Prolog

- Prolog is a ‘typeless’ language with a very simple syntax.
- Prolog is a logical programming language.
- Prolog uses
  - Horn clauses
  - implements resolution
    » using a “depth first” strategy
    » unification
- Atoms, lists and records are the data structures.
Declarative Languages

- Also known as *logic programming*
- Examples: Prolog, SQL, Datalog.
- Typical of database systems and artificial intelligence.

- Declarative specifications: Specify *what* you want, not *how* to compute it.
- E.g. Find X and Y such that
  
  \[ 3X + 2Y = 1 \]
  \[ X - Y = 4 \]
Classical First-Order Logic

- simplest form of logical statements is an atomic formula. e.g.
  - man(tom)
  - woman(mary)
  - married(tom,mary)
- More complex formulas can be built up using logical connectives: $\land, \lor, \neg, \forall X, \exists X$. 
Examples of First Order Logic

smart(tom) w dumb(tom)
smart(tom) w tall(tom)
5dumb(tom)
ôX married(tom, X)
úX loves(tom, X)
ôX [married(tom, X) v female(X) v human(X)]
rich(tom) w5 smart(tom)
This implies that if tom is smart, then he must be rich. So, we often write this as
rich(tom) 7 smart(tom)
In general, P 7 Q and Q 6 P are abbreviations for
P w5 Q
Examples:
ú X[((person(X) v smart(X)) 6 rich(X)]
ô X mother(john,X)
ô X [mother(john,X) v ú Y[mother(john,Y) 6 Y=X] ]
Horn Rules

- Logic programming is based on formulas called Horn rules. These have the form
  \[ \forall x_1, \ldots, x_k [A \leftarrow B_1 \land B_2 \land \cdots \land B_j] \]

- Examples:
  - \[ X, Y[A(X) \land B(X,Y) \lor C(Y)] \]
  - \[ X[A(X) \land B(X)] \]
  - \[ X[A(X,d) \land B(X,e)] \]
  - \[ A(c,d) \land B(d,e) \]
  - \[ X A(X) \]
  - \[ X A(X,d) \]
  - \[ A(c,d) \]

- Note that atomic formulas are also Horn rules, often called facts.

- A set of Horn rules is called a Logic Program.
• Logic programming is based on a simple idea: From rules and facts, derive more facts.

• Example 1. Given the facts A, B, C, D, and the rules:
  1. E 7 A v B
  2. F 7 C v D
  3. G 7 E v F

  From 1, derive E; from 2, derive F; from 3, derive G.
Logical Inference

• Example 2: Given these facts:
  man(plato)
  man(socrates)
and this rule:
  \( \forall X [\text{man}(X) \land \text{mortal}(X)] \)
derive:
  mortal(plato), mortal(socrates).
Recursive Inference

- Example, given
  \[ \text{ú X[mortal(X) \& mortal(son_of(X))]} \]
mortal(plato)
derive:
mortal(son_of(plato))
  (using X=plato)
mortal(son_of(son_of(plato)))
    (using X=son_of(plato))
mortal(son_of(son_of(son_of(plato))))
      (using X=son_of(son_of(plato)))
Horn rules correspond to programs, and a form of Horn inference corresponds to execution.

For example, consider the rule:
\[ \text{ú X,Y p(X) 7 q(X,Y) v r(X,Y) v s(X,Y)} \]
This rule can be interpreted as a program where
- \( p \) is the program name,
- \( q, r, s \) are subroutine names,
- \( X \) is a parameter of the program, and
- \( Y \) is a local variable.
Non-Horn Formulas

• The following formulas are not Horn:
  A 6 5 B
  A wB
  A wB 6 C
  ÕX[A(X) 7 B(X)]
  A 7 (B 7 C)
  úX[flag(X) 6 [red(X) wwhite(X)]]
  ú X ŕ Y[wife(X) 6 married(X,Y)]
Non-Horn Inference

• Non-Horn inference is more complex than with Horn formulas alone. Example:
  A 7 B
  A 7 C
  B wC (non-Horn)

We can infer A, but only by doing case analysis either B or C is true.
  if B then A
  if C then A

Therefore, A.

• Non-Horn formulas do not correspond to programs, and non-Horn inference does not correspond to execution.
Logical Equivalence

- Many non-Horn formulas can be put into Horn form by using either:
  - logical equivalence
  - Skolemization (a subject for a later course).
- Example of logical equivalence:
  \[ 5 \mathit{A} \ 7 \ 5 \mathit{B} / \ 5 \mathit{A} \ \mathit{w5} \ (5 \mathit{B}) \]
  \[ / \ 5 \mathit{A} \ \mathit{wB} \]
  \[ / \ \mathit{B} \ \mathit{w5} \mathit{A} \]
  \[ / \ \mathit{B} \ 7 \mathit{A} \quad \text{(Horn)} \]
Logical Laws

5 5 A / A
5(A wB) / 5 A v 5 B
A w(B v C) / (A wB) v (A wC)
A 7 B / A w5 B

• Example using logical equivalence and laws:
  A 7 (B wC) / A w5 (B w C)
    / A w(5B) v (5 C)
    / (A w5B) v (A w5 C)
    / (A 7 B) v (A 7 C) (Horn)
Non-convertible Formulas

• In general, rules of the following form cannot be converted into Horn form:

$$\forall x[(A_1 \lor \ldots \lor A_n) \leftarrow (B_1 \land \ldots \land B_m)]$$

For example,

(A wB) 7 (C v D)
(A wB) 7 C
(A wB)

i.e., if it is possible to infer a non-trivial disjunction from a set of formulas, then the set is inherently non-Horn.
• Syntax is Horn clauses of *terms*.
• Proof process involves *SLD Resolution*
  – unification + substitution: pattern matching between terms + binding unresolved variables as needed.
  – automatic backtracking: if one attempt fails, try again until all search paths are exhausted.

SLD stands for Selecting a literal, using a Linear strategy, restricted to Definite clauses. The name SLD was coined by researchers in automatic theorem proving before the birth of logic programming.
For convenience, we don’t write the universal quantifiers, so a rule:
\[ \forall X \ [p(X) \lor q(X) \lor r(X)] \]
is written as
\[ p(X) \lor q(X), r(X). \]

We also use the Prolog conventions:
- *variables* begin with upper case (A, B, X, Y, Big, Small, ACE)
- *constants* begin with lower case (a, b, x, y, plato, aristotle)
Prolog Syntax

< fact >  6 < term > .

< rule >  6 < term > :- < terms > .

< query >  6 < terms > .

< term >  6 < number >  |  < atom >

|  < variable >  |  < atom > ( < terms > )

< terms >  6 < term >  |  < term > , < terms >
Syntax

• Integers: base 10

• atoms: user defined, supplied
  – name starts with lower case: john, student2

• Variables
  – begin with upper case: Who, X
  – ‘_’ can be used in place of variable name

• Structures
  – student(ali, freshman, 194).

• Lists
  – [x, y, Z ]
  – [ Head | Tail ]
  » syntactic sugar for . ( Head, Tail )
  – [ ]
/* list of facts in prolog, stored in an ascii file, ‘family.pl’*/
mother(mary, ann).
mother(mary, joe).
mother(sue, mary).

father(mike, ann).
father(mike, joe).

grandparent(sue, ann).

/* reading the facts from a file */
1?- consult (‘family.pl’).
family.pl compiled, 0.00 sec, 828 bytes
• Comments are either bound by “/*”, “*/” or any characters following the “%”.

• Structures are just relationships. There are no inputs or outputs for the variables of the structures.

• The swipl documentation of the built-in predicates does indicate how the variables should be used.
  
  pred(+var1, -var2, +var3).
  
  + indicates input variable
  
  - indicates output variable

• You can also consult a file with a “pl” extension by

  ?- [family].
/* Prolog the order of the facts and rules is the order it is searched in */
/* Variation from pure logic model */

2 ?- father(X, Y).

X = mike /* italics represents computer output */
Y = ann ; /* I type ‘;’ to continue searching the data base */

X = mike
Y = joe ;

no

3 ?- father(X, joe).

X = mike ;

no
/* Rules */
parent( X , Y ) :- mother( X , Y ). /* If mother( X ,Y ) then parent( X ,Y ) */
parent( X , Y ) :- father( X , Y ).

/* If the facts are later then   grandparent(sue, ann).  redundant */
/* if parent( X ,Y ) and parent(Y,Z ) then grandparent( X ,Z ). */
grandparent( X , Z ) :- parent( X , Y ),parent(Y, Z ).
‘or’

parent( X, Y ) :- mother( X, Y ); father( X, Y ).

has same effect as:
parent( X, Y ) :- mother( X, Y ).
parent( X, Y ) :- father( X, Y ).
mother(mary, ann).
mother(mary, joe).
mother(sue, marY ).
father(mike, ann).
father(mike, joe).

parent( X, Y ) :- mother( X, Y ).
parent( X, Y ) :- father( X, Y ).

?- parent( X, joe).
X = mary
yes
?- parent(X, ann), parent(X, joe).
X = mary;
X = mike
yes

?- grandparent(sue, Y).
Y = ann;
Y = joe
yes
/* specification of factorial n! */
factorial(0,1).
factorial(N, M):– N1 is N – 1, factorial (N1, M1), M is N*M1.

?- factorial (2, X).
M = X, N = 2, N1 = 1

?- factorial (1, X1).
(X1 is the M1 above)
M = X1, N = 1, N1 = 0

?- factorial (0, X2).
(X2 is the M1 above)
X2 = 1

factorial (0, 1).  /* succeeds */
/* after the first rule succeeds, the second rule is not used */
r1:  addToSet( X, L, L ) :- member( X, L ).

?- addToSet( a, [b,c], O ).
Recursion in Prolog

• trivial, or boundary cases
• ‘general’ cases where the solution is constructed from solutions of (simpler) version of the original problem itself.

• What is the length of a list?
• THINK:
The length of a list, \([e | \text{Tail}]\), is \(1 + \) the length of Tail

• What is the boundary condition?
  – The list \([\ ]\) is the boundary. The length of \([\ ]\) is 0.

• Where do we store the value of the length?--accumulator--

\[
\text{length}([\ ], 0).
\]
\[
\text{length}([H \mid T], N) :- \text{length}(T, Nx), N \text{ is } Nx + 1
\]
Recursion

myleNGTH([ ], 0).
myleNGTH([X | Y], N):-myleNGTH(Y, Nx), N is Nx+1.

? – mylength([1, 7, 9], X ).
X = 3

? - mylength(jim, X ).
no

? - mylength(Jim, X ).
Jim = [ ]
X = 0
Recursion

mymember( X , [X | _ ] ).
mymember( X , [ _ | Z ] ) :-  mymember( X , Z ).

% equivalently: However swipl will give a warning
% Singleton variables : Y W
mymember( X , [X | Y] ).

1?-mymember(a, [b, c, 6] ).
   no
2? – mymember(a, [b, a, 6] ).
   yes
   X = b;
   X = c;
   X = 6;
   no