Benjamin Fenster
CS-251 (Wu)
18 December 2005

Logical Programming

I. Introduction to Logical Programming
   a. Example
      i. Suppose you want to define your ancestors. Your parents are your ancestors, and your parents' ancestors are your ancestors.
      ii. ancestor(X, Y) :- parent_child(X, Y).
         1. This is a predicate ("isAncestor")
         2. So X is an ancestor of Y if X is the parent of Y.
         3. This is the base case
      iii. ancestor(X, Y): - parent_child(X, Z), ancestor(Z, Y)
         1. Here X is Y's ancestor if X is Z's parent and Z is Y's ancestor
         2. There's the recursive case
      iv. Any uppercase letter begins a variable name
      v. All we've done is define the problem's logic. We haven't said anything about how to solve it.
      vi. There's lots of recursion in Prolog, so get used to it.
      vii. It's symbolic too (not numeric, symbolic)
   b. Why use Prolog?
      i. Logical programming is one of the four programming paradigms (the others being procedural, functional, and object-oriented)
      ii. Theoretically you don't have to worry about data types, though in small Prolog systems it's more efficient to declare types.
   c. Propositional Calculus
      i. Have single propositions, not variables
      ii. Have and, or, not, implication (=>), and =
      iii. See [MATH-054] for details on all that.
      iv. Compose sentences with true/false/p/symbols
      v. Legal sentences are well-formed (WFS).
      vi. In p => q, p is the premise / antecedent and q is the conclusion / consequent
   d. Predicates Calculus
      i. We can now break the proposition into pieces so we can use variables
      ii. "Ben's car has 5 doors" becomes car_door(Ben, 5)

II. Introduction to Turbo Prolog
   a. Every data object is a Term
   b. Atomic Terms (constants, really)
      i. characters, integers, reals, strings,
      ii. symbols (like <u>activity</u> or <u>person</u>)
      iii. files (we're not really concerned with this in CS-251)
   c. Function Terms
      i. <functor> {<term1>{<term>})}
      ii. Basically we're saying name(args)
      iii. The number of arguments is the airity of the function
      iv. We usually write <functor>/<airity>
      v. So: `grade_attained/2`
   d. Composite Terms
      i. owns(xindong, book("Title", 1995))
      ii. Just like it sounds: create one term out of multiple pieces
   e. Variables
      i. One of the most difficult terms in Prolog!
      ii. Each variable starts with an uppercase letter or _, where _ works like it does in OCaml to mean "I don't care about this value" (see [CS-103])
      iii. When you start a symbol with a lowercase letter it refers to a particular instance (like `xindong`)
   f. Predicates: Basically the same as functions, but we expect variables to be involved

g. Horn Clauses
    i. LHS :- RHS
    ii. LHS (head) is a single predicate called the consequent (it's what you're trying to define)
    iii. RHS (tail) is zero or more predicates separated by commas
    iv. When the body has zero predicates it's called a "fact"
    v. When there are body predicates it's called a "rule"
    vi. Think of a fact as a rule with "true" for its tail, so `person(name) :- true` means that anybody with a name is a person, basically.

h. Example
    i. Code

```
can_study_advanced_ai :- has_studied_ai.
can_study_advanced_ai :- department_head_says_okay.
will_study_advanced_ai :- can_study_advanced_ai,
                          has_enrolled
```

    ii. The first two form a logical or.
    iii. The comma in the third forms a logical and.

i. Queries
    i. Once you've given all the rules, pose queries against them
    ii. Given a ?- prompt in the dialog window
    iii. Issue a query in the same form as facts

III. Backtracking

a. One of the most difficult parts of logical programming is backtracking.
b. It's used when there's more than one version of a clause
c. Example
    i. Suppose we assert `department_head_says_okay` in the earlier example and then ask `?- can_study_adv_ai`
    ii. The Prolog system first checks has_studied_ai but that fails. It then backtracks to department_head_says_okay and that succeeds.
d. It's possible to use a semicolon to achieve the same effect (A :- B; C) but it's preferable to use two separate rules (A :- B   A :- C)
e. Now that the Prolog system will backtrack, it's possible to get more than one answer. For example, if both `has_studied_ai` and `department_head_says_okay` are given as facts, you should get `yes` as the answer twice. The prolog system will prompt with `yes ?` and you can use a `;` to ask for the next answer.

IV. Variables

a. The same variable name in different clauses are completely independent
b. Example
    i. Code

```
has_studied(Everyone, cs021)
can_study(Anyone, Anything) :- has_studied(Anyone,
prereq_to(Anything))
```

    ii. Note that cs021 is not a variable.

c. Unification
    i. Variables are initially unbound and need to get bound at some point. That process is called unification
    ii. Two variables unify if:
        1. They are identical
        2. They are both functions with the same functor and their arguments pairwise unify (which means just what you'd think)
        3. One is a variable and one is a value, in which case we bind one to the other (or if both are variables then the value of one to the other)
    iii. We don't allow second order logic (so f(a, b) = Anyfunc(a, b) won't unify)

V. Miscellaneous Items

a. I/O: done with readln(X), write("The value of X is", X).

b. Not: Done as `x <> y` or `not(x = y)`
c. Anonymous Variables: Use just _ when you completely don't care about the value. Use _name if you don't care but still wish to name it.
d. Arithmetic
   i. Evaluation is caused by the = predicate
   ii. The RHS can contain +, -, *, /, mod but must be evaluable (no unbound variables are allowed)
   iii. The LHS must be on evariable or one value only
   iv. Example: Greatest Common Divisor
      1. Code

```
gcd(X, X, X).  /* gcd of X and itself is X */
gcd(X, Y, GCD) :- X < Y,
                  Diff = Y – X
                  gcd(X, Diff, GCD).
gcd(X, Y, GCD) :- gcd(Y, X, GCD).
```

VI. Lists
a. domains: sometime = integer* (the Klene star)
b. Works just like OCaml in how the head and tail separate. See [CS-103] for details.
c. member(Element, [Element|_]). member(Element, [_, Tail]) :- member(Element, Tail).
d. Append
   i. Code

```
append([], List, List).
append([Head|Tail], List, [Head|NewList]) :-
      append(Tail, List, NewList).
```

   ii. Strip off elements on the way into the recursion, then reattach them at the beginning on the way back out.
e. Length
   i. Code

```
length([], 0).
length([Head|Tail], Length) :-
        length(Tail, TLength),
        Length is TLength + 1.
```

   ii. is works like = but is more flexible
f. Interesting Functions
   i. member(Element, List) :- append(_, [Element|_], List).
   ii. If Element is a member of List then that means it's the head of some list, appended to the end of some other list. (So it's in the middle somewhere.)
   iii. sublist(Sublist, List) :- append(_, BackHalf, List), append(Sublist, _, BackHalf).
   iv. Think about that second condition first (it's easier to understand).
   v. Take the sublist and stick stuff after it (possibly "empty" stuff, remember) to get the Tail of the main list.
   vi. Then append that whole thing to another (possibly empty) list which is the beginning.
   vii. Thus we've added the necessary stuff before and after the Sublist and come up with the original List so the predicate is true

VII. Negation by Failure
a. \+ tests if its argument fails
b. It's based on the Closed World Assumption
c. clever(X) :- \+(stupid (X))
d. So the lack of information supporting stupid(X) must mean X is clever.
e. This requires finite failure. If you have infinite recursion somewhere, negation by failure won't work.
f. You also can't use this with unbound variables. not (not (p)) is NOT always p

VIII. The Cut
a. Cutting means: Once you've gotten here, don't worry about any other alternative
b. Whether it's true or false at this point, drop the backtracking pointer and move on.

    c. Done with !
    d. member(X, [X|_]) :- !.
    e. This can change the declarative meaning though (called a red cut in that case). Green cuts don't change the meaning, just change performance considerations by avoiding unnecessary backtracking.

IX. Database Facts
    a. Store database information as facts. You can assert (add) and retract (remove) facts for the head predicate symbols you've identified
    b. asserta(X) adds X above similar facts
    c. assertz(X) adds X below similar facts
    d. retract(X) removes everything that unifies with X

X. Findall
    a. Suppose you have a database of facts like instructor(xindong, ai)
    b. findall(Subject, instructor(Person, Subject), Subject_List)
    c. This will fill Subject_List with all the Subject values that match that instructor() clause.
    d. bagof
        i. Not implemented in Visual or Turbo Prolog
        ii. bagof(Variable, Query, ListOfSolutions)
        iii. Bags are organized by the other variables listed
    e. setoff
        i. Removes duplicates
        ii. Still groups by other variables