Programming Languages Workshop CS 67420

Lecture 7

Huji, Spring 2016

Dr. Arie Schlesinger

Prevs summary

Kb's

- A fact declares that something is true.
- A rule states conditions for something to be true.
- *Programs* are kb's of facts and rules.
- Queries are submitted to the system to initiate a computation.

Summary: Answering Queries

- When answering a query, Prolog tries to prove that the corresponding goal is satisfiable (can be made true).

- This is done using the rules and facts given in a program.

- A goal is executed by matching it with the *first possible* fact or head of a rule. In the latter case, the rule's body becomes the new goal.

- The variable instantiations made during matching are *carried along throughout the computation and reported at the end*.

- Only the anonymous variable _ can be instantiated differently whenever it occurs.

Terms

- Prolog terms are either
  - numbers,
  - atoms,
  - variables, or
  - compound terms.

- Atoms: start with a lower case letter, or are enclosed in single quotes:

  `moshe, abC, x_3, ' Can you whistle ?'`

- Vars: start with a capital letter, or with an underscore:

  `X1, Epsilon, _var7, LastVar, _`

Matching

- Two terms match if:

- are *identical*, or
- can be made *identical* by substituting *their vars* with suitable *ground terms.*

- Use =/2 predicate to ask Prolog if 2 terms match:

```
?- loves(iosi, ety) = loves(iosi, X).
X = ety
```

Prolog displays the values that enabled the match.

Matching compound terms

```
?- f(c, k(X, Y)) = f(X, Z), Z = k(W, p(X)).
X = c
Y = p(c)
Z = k(c, p(c))
W = c
true

?- k(X, 2, 2) = k(1, Y, X).
false
```

Using = (unification) yields more flexible code than using == (testing for equality).

```
?- X=1+1.
X = 1+1.

?- X==1+1.
false.
```

=/2

=/2 succeeds when the two terms are unified.

A = 2 or 2 = A ,are the same thing, a goal to unify A with 2.

```
?- 2=2.
true.
```

```
?- A=2.
A = 2.
```

==/2

The ==/2 "operator” succeeds only if the two terms are already identical without further unification.

A == 2 is true only if the variable A had **previously** been assigned the value 2.

```
?- 2==2.
true.

?- A==2.
false.

?- B=2, B==2.
B = 2.
```

\=

\= means the two terms cannot be unified : unification fails.

"not unified" does not result in any unification between terms.

```
?- 2\=2.
false.

?- 2\=3.
true.

?- A\=2.
false.
```

\==

\== means the two terms are not identical. Here also no unification takes place even if this succeeds.

```
?- 2\==2.
false.

?- 2\==3.
True.

?- A\==3.
true.

?- A\==3,A = 5.
A = 5.
```

Matching with the Anonymous Variable

- The variable _ (underscore) is called the *anonymous* variable.

- Every occurrence of _ may represent a diff var.

```
?-p(_, 2, 2) = p(1, Y, _).
Y=2
true
```

_ first it unifies with 1,

then with 2,

so all these changing values, are not printed

Matching Queries

- If a goal matches with a *fact*, then it is satisfied.

- If a goal matches the *head of a rule*, then it is satisfied if the goal represented by the rule's body is satisfied.

- If a goal consists of several sub-goals separated by commas, then it is satisfied if all its sub-goals are satisfied.

- When trying to satisfy goals with built-in predicates such as write/1, Prolog also performs the action associated with it (e.g., writing something on the screen).

```
?-X=Y,Y=2,write(X).
2
X = Y, Y = 2.
```

Built-in Predicates

- Read and compile a program file:

```
?- consult('aProg.pl').
true
```

- Displaying terms on the screen :

```
?- write('The blue is also sky'), nl.
The blue is also sky
true
```

nl is a new line

what helps readability

- meaningful names
- spaces, indentation
- one clause per line
- Coments – long, short:

```
/* a long comment,…
                    ... */

uncle(X,Y) :- brother(X, Z), % some males..(a short comment.)
parent(Z,Y).
```

Some predicates

examples

socrates

```
All men are mortal.
Socrates is a man.
Hence, Socrates is mortal.
```

In Prolog:

```
mortal(X):-man(X).
man(socrates).

?-mortal(socrates).
true
```

Family business

```
father(X,Y) :- parent(X,Y), male(X).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

paternalgrandfather(X,Y) :- father(X,Z), father(Z,Y).

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

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), descend(Z,Y).
```

lists

sequences of things

a special kind of data structure

Lists in Prolog

- Example :

```
[ship, plane, car, bike]
```

-square brackets, commas between elements

-The empty list is written as: [ ].

-elements : *any* Prolog terms (vars, and other lists too).

```
[T, _X, [], g(X,Z,y), _ ,8, [D,d,c], reads(iosi,Book)]
```

The built-in bar | operator

- Prolog has a special built-in operator | (bar), which can be used to *decompose a list into its head and tail*
- The | operator is a main tool for writing list manipulation predicates :

```
?- [Head | Tail] = [a,b,c,d].
Head = a,
Tail = [b, c, d].
```

?- [H|T]==[a,b,c,d]. % equality check does'nt work in decomposition business

false.

```
?- [Head|Tail] = [].
false.
?- [X,Y|Tail] = [a,b,c,d,e,f].
X = a,
Y = b,
Tail = [c, d, e, f].
```

more list partitions

- If a bar | is put just before the *last term in a list*, this means that this last term denotes a sub-list.

?- [1,2,3,4]=[1|X].

X = [2, 3, 4].

?- [1,2,3,4]=[1,2|X].

X = [3, 4].

?-[1,2,3,4]=[1,2,3|X].

X = [4].

More examples

```
Remember cons from racket? – a parallel view
(cons Head Tail)   …   [Head|Tail]
(cons a Tail) … [a|Tail]          % Tail is a sublist
(cons a (cons b Tail)) … [a,b|Tail]
(cons a (cons b (cons c []))) … [a,b,c]

?- [ship, plane, bus, car] = [Head | Tail].
Head = ship
Tail = [plane, bus, car]

?- [1,2,3,4]=[1,Y,3|X].% extract from specific locations
Y = 2,
X = [4].% right to the bar lives a sublist
```

more Head and Tail

- Another example:

```
?- [bike] = [X | Y].
X = bike
Y = []

?- [[iosi],[ety]] = [X|Y].
X = [iosi],
Y = [[ety]].
```

Head and tail of empty list

- The empty list has *neither* a head nor a tail

- In Prolog, [ ] is a special simple list *without any internal structure*

- The empty list is important for writing recursive predicates, and for list processing in Prolog

Using anonymous vars

Using anonymous variables

- A simple way of obtaining only the information we want:

The 3rd and the 5th elements:

```
?- [_,_,X,_,Y|_]=[a,b,c,d,e,f,g,h].
X = c,  % the 3rd element
Y = e.  % the 5th element
```

Examples

- Extract the second element from a given list:

```
?- [a, b, c, d, e] = [_, X | _].
X=b
```

- Make sure the first element is a 1 and get the rest elements after the second element "packed in a list":

```
?- MyList = [1, 2, 3, 4, 5], MyList = [1, _ | Rest].
MyList = [1, 2, 3, 4, 5]
Rest = [3, 4, 5]
```

List predicates examples

the recursive machinery of my_append

- base case: when one of the lists is empty.

- Recursion:
- In every step we *take off the head,* and recall the predicate, with the (shorter) tail. Done until the base case is reached.

```
my_append([], List, List).
my_append([Elem|List1], List2, [Elem|List3]) :-
                              my_append(List1, List2, List3).
```

Appending Lists

```
?- my_append ([1,2], [3,4], L).
L = [1,2,3,4]
```

Interesting uses:

```
?- my_append([1,2],L,[1,2,3,4]).
L = [3,4]

?- my_append(L,[1,2],[1,2,3,4]).
false.

?- my_append(L,[3],[1,2,3,4]).
False.
?- my_append(L,[3,4],[1,2,3,4]).
L = [1, 2] .
```

More interesting uses: decomposing lists –" חלקי חילוף "

```
?- my_append(L1, L2, [a, b, c]).
L1 = []
L2 = [a, b, c] ;

L1 = [a]
L2 = [b, c] ;

L1 = [a, b]
L2 = [c] ;

L1 = [a, b, c]
L2 = [] ;
false
```

member – חברי מועדון

```
member(X, [X|_]).                              % true if X is the Head
member(X, [_|Tail]) :- member(X, Tail).        % true if X is in the Tail
```

- One liner with or:

```
member(X, [Y|T]) :- X = Y; member(X, T).
```

member & bar “adventures”

```
?- member(a,L).% list L unknown yet
```

L = [a|_G1805]

L = [_G1804, a|_G1808]

L = [_G1804, _G1807, a|_G1811]

L = [_G1804, _G1807, _G1810, a|_G1814]

L = [_G1804, _G1807, _G1810, _G1813, a|_G1817]

L = [_G1804, _G1807, _G1810, _G1813, _G1816, a|_G1820]

L = [_G1804, _G1807, _G1810, _G1813, _G1816, _G1819, a|_G1823]

L = [_G1804, _G1807, _G1810, _G1813, _G1816, _G1819, _G1822, a|_G1826]

L = [_G1804, _G1807, _G1810, _G1813, _G1816, _G1819, _G1822, _G1825, a|...] .

reverse/3 a list

```
reverse([],X,X).
reverse([H | T], X, Acc) :- reverse(T, X,[H | Acc]).

?- reverse([1,2],X,Y).
X = [2, 1|Y].

?- Y=[],reverse([1,2],X,Y).
Y = [],
X = [2, 1].

?- Y=[7],reverse([1,2],X,Y).
Y = [7],
X = [2, 1, 7].

?- Y=[7],reverse([1,2],X,[8]).
Y = [7],
X = [2, 1, 8].
```

2 more reverse runs

```
?- reverse([1,2],X,[]).
X = [2, 1].

?- reverse([1,2,3],X,[]).
X = [3, 2, 1].
```

Arithmetic in Prolog

Expressions

successor: a way to describe the natural numbers:

```
natNumber(0).              % 0 is a natNumber.
natNumber(succ(X)):-
             natNumber(X). % If X is a natNumber, then so is succ(X).

?- natNumber(succ(succ(succ(succ(0))))).
true
?- natNumber(X).
X = 0 ;
X = succ(0);
X = succ(succ(0));
X = succ(succ(succ(0)))
X = succ(succ(succ(succ(0))))
X = succ(succ(succ(succ(succ(0)))))
X = succ(succ(succ(succ(succ(succ(0))))))
X = succ(succ(succ(succ(succ(succ(succ(0)))))))
X = succ(succ(succ(succ(succ(succ(succ(succ(0))))))))
X = succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))
X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ( . . . ))))))))))
```

addition with successors

```
add(0,X,X).                          % base clause

add(succ(X),Y,succ(Z)):-add(X,Y,Z).  % recursive clause

Try : -, *, **
```

Arithmetic operators

- Arithmetic expressions in Prolog are *just terms*, which are not evaluated *automatically*. *("Everything is a term in Prolog")*

- (Remember quoted lists in Racket).

- So, `2 + 3` is just the same as `+(2,3)`, which does *nothing* on its own.

- It is the responsibility of individual predicates to *evaluate* those terms.

- Several built-in predicates do *implicit evaluation*, like :

- arithmetic comparison operators like =:=, (there are more..)
- and is.

=:= *operator*

=:= *evaluates* both args on (both sides) and *compares* their values,

```
?- 2=:=1+1.
true.
?- 3-1=:=7-5.
true.

?-X = 6, X =:= 2+3.
false.

?- X=1+4, X=:=2+3. % evaluates 1+4 only at =:= not in X=1+4
X = 1+4.
```

?- 1+4=:=2+3.

true.

The is/2 operator

`is` accepts and evaluates only its *right* arg as an arith expression.

`is` *left* arg has to be an *atom*, either a *numeric* constant (which is then compared to the result of the evaluation of the right operand), or a *variable*.

- If it is a *bound* variable, its value has to be numeric and is compared to the right operand as in the former case.

```
?- X=6, X is 7-1.
X = 6.
```

- If it is an *unbound* variable, the result of the evaluation of the right operand *will be bounded* to that variable.

```
?- X is 5/6-2.                % start with an unbounded X.
X = -1.1666666666666665.
```

`is` is often used in this latter case, to bind variables, the prolog assignment

More is/2-op examples

```
?- X is 2+3.
X = 5 ;

?- X is 12*-4.
X = -48 ;

?- X is 2*4.
X = 8 ;

?- X is 2/3.
X = 0.6666666666666666.
```

Errors of "habit"

```
?- 2+3 is 5.  % no calculations on the left side.
false

?- 5 is 2+3.  % is calculates on the right side and unifies with 5.
true.

?- 5 is 2+X.  % unification is not wizardry
ERROR: Arguments are not sufficiently instantiated
```

But:

```
?- X = 3, 5 is 2+X.     % this will work
X = 3.

?- X = 3, 5 is 1+X.     % works but is false
false.
```

is/2 or =:=/2

- To test if a number N is even, we could use both operators:

```
0 is N mod 2  % true if N is even
0 =:= N mod 2 % true as above
```

But if you want at the same time, *to capture the result* of the operation, you can only use the first variant. If X is unbound, then:

```
X is N mod 2  % X will be 0 if N is even
X =:= N mod 2 % will errorize!! with argument/instantiation error! (of X)
```

For just *comparison of values of arith expressions*, use =:=.

To *capture the result* of an evaluation, use is.

is/2 or =:=/2 or ==/2

```
?- N=8, X =:= N mod 2 .
ERROR: =:=/2: Arguments are not sufficiently instantiated (X..)

?- N=8, X is N mod 2. % X is bounded to mod result
N = 8,
X = 0.
```

=:= is a comparison operator.

A1 `=:=` A2 succeeds if *values* of expressions A1 and A2 are equal.

A1 **==** A2 succeeds if *terms* A1 and A2 are *identical*;

Checking if a num is odd

Using is/2 :

```
odd(Num) :- 1 is Num mod 2.
```

Using =:=/2 :

```
odd(Num) :- Num mod 2 =:= 1.
```

More cautious? check first if Num is integer:

```
odd(Num) :- integer(Num), 1 is Num mod 2.
```

=/2 does not evaluate

```
No arith evaluations are done here just term matching :
?- 2+3 = +(2,3). % 2+3 is a valid Prolog term
true             % as a match of operators not of arith values

?- 2+3 = +(3,2). % args not in the right places ? False!
false

?- 2+3 = 5. % no wander..
False
?- X is 2 + 3. % is does it
X = 5
```

int vs float battles of honour

With is we can open an arithmetic center:

```
?- X is 2 * 4 + 3 * -7, Y is X / 5, Z is Y/X.
X = -13,
Y = -2.6,
Z = 0.2.

?- X is 1.2 + 2.8, X = 4.
false.

?- X is 1.2 + 2.8, X = 4.0.
X = 4.0.
```

- Some systems will try to instantiate X with the `int` 4,
- others with the `float` 4.0.

Some built-in arith functions

- Examples:

```
?- X is max(12, 11).
X = 12.

?- X is sqrt(11.23) * 1.2.
X = 4.021343059227849.


?- X is (123 mod 11) ** -1.2.
X = 0.43527528164806206.
```

More functions in the Manual

Built-in predicates for comparing vals of arith exprs, all evaluate their L,R args, as does =:=

```
?- 7 > 2.
true
?- 7+2 > 8-2.
true
?- 7 =< 2.
false
?- 7 =:= 2.
false
?- 7 =\= 2. % not equal
true
?- 7 >= 2.
true
```

Find the max number in a list

- It takes 3 args: a list, *Max* (a var for the max) in the list, and *Res* (var for the ans):
- Max gets updated with the highest number met so far.

```
findMax([H|T],Max,Res):-
                     H > Max,
                     findMax(T,H,Res).
findMax([H|T],Max,Res):-
                     H =< Max,
                     findMax(T,Max,Res).
findMax([],Max,Max).
```

getMax/2 wrapper for findMax/3

```
?- findMax([5,2,6,1], 0, Res).
Res = 6.

?- Max=-1, findMax([5,2,6,1], Max, Res).
Max = -1
Res = 6.
```

Wrapper getMax/2:

```
getMax([H|T],Res):- findMax(T,H,Res).
```

Use:

```
?- getMax([1,5,2,7,2,99],X).
X = 99.
```

Examples

- Recall the difference between matching and arithmetic evaluation:

```
?- 3 + 5 = 5 + 3.
false
?- 3 + 5 =:= 5 + 3.
true

Different precedencies :
?- 2 + 3 * 4 =:= (2 + 3) * 4.
false
?- 2 + 3 * 4 =:= 2 + (3 * 4).
true
```

Summary: Arith in Prolog

- For logical pattern matching use **`=`**,

- for arithmetic evaluation use the **`is op`**.

- A range of built-in arithmetic functions is available

(some are written as *infix* operators, e.g., `+`).

- Arithmetic expressions can be *compared* using arithmetic relations such as

  < or **`=:=`** (i.e., not using the is-operator)

Length of a list

Using the arithmetic operator is:

```
len([],0).
len([H|T],N) :- len(T,M), N is M+1.
```

The second clause can also be

```
% who cares what H is, after counting it here..
len([_|T],N) :- len(T,M), N is M+1.
```

```
?- len([11,12,-33,-4],N).
N=4
```