Today

- Prolog II

Principles of Programming Languages IX

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.dgp.toronto.edu/~wael/324.html

Prolog: unification operators

- unify with operator: $X = Y$
  - Semantically: unifiable test
  - Succeeds as long as $X$ and $Y$ can be unified
  - $X$ may or may not be instantiated, $Y$ may or may not be instantiated
  - As a side effect, $X$ and $Y$ become bound together (refer to the same object)
  - E.g.:
    - $\text{X} = \text{Joe}$
    - $\text{Y} = \text{Joe}$
    - $\text{X} = \text{Y}$
    - $\text{X} = \text{Y}$

- equals does not unify with operator: $X \neq Y$
  - Semantically: not-unifiable test
  - Succeeds as long as $X$ and $Y$ cannot be unified
  - Both $X$ and $Y$ must be instantiated but may have uninstantiated elements.
  - No side effects
  - E.g.:
    - $\text{X} \neq \text{Y}$
    - $\text{Y} \neq \text{Y}$

- == is already instantiated to operator: $X == Y$
  - Semantically: identical test
  - Succeeds as long as $X$ and $Y$ are already instantiated to the same object
  - Any variable inside $X$ and $Y$ must be the same
  - No side effects
  - E.g.:
    - $\text{X} == \text{2 + 2}$
    - $\text{Y} == \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$

- =:= is already instantiated to operator: $X =:= Y$
  - Semantically: identical test after evaluating terms
  - E.g.:
    - $\text{X} =:= \text{2 + 2}$
    - $\text{Y} =:= \text{2 + 2}$
### Prolog: unification operators

- **\( \text{==} \)**: not already instantiated to operator
  - Semantically: not-identical test
  - Succeeds as long as X and Y are not already instantiated to the same object
  - No side effects
  - E.g.
    - `? A \== hello. Yes`
    - `? a(b,X,c) \== a(b,Y,c). Yes`
    - `? 1 + 2 \== 3 Yes`

- **\( == \)**: already instantiated to operator
  - Semantically: not-identical test after evaluating terms
  - E.g.
    - `? 4 == 2 + 2 No`

### Prolog: lists

- A sequence of terms of the form
  \[ [t_1, t_2, t_3, \ldots, t_n] \]
  where term \( t_i \) is the \( i \)th element of the list

- \([\,]\) is the ‘empty list’. It is an atom not a list.

- Example:
  - `[a, b, c, [d, e, [[f]]]]`
  - A list with 4 elements: a, b, c, and a list with 4 elements d, e, an empty list, and f
  - Prolog supports Scheme-style nested lists

- Can break apart lists using “\( | \)” into \([\text{Head} | \text{Tail}]\) where Head is the first item as an object and Tail is the rest of the list (as a list)
  - E.g.
    - `?- [H | T] = [a, b, c].
      H = a
      T = [b, c]

- You can also use the same notation “\( | \)” to construct lists:
  - E.g.
    - `?- L = [a | [b, c]].
      L = [a, b, c]`

### Prolog: unification & lists

- **Other arithmetic operators**:
  - Add, subtract, multiply, division
  - Integer division
  - Modulus
  - Comparison after evaluation
  - E.g.
    - `\|\| X \equiv 2 + 2 No`

- **is** operator: \( X \equiv \text{Expr} \equiv \text{is}(X,\text{Expr}) \)
  - Semantically: evaluate second term and test if it is equal to \( X \)
  - Succeeds a long as \( X \) and the arithmetic evaluation of \( \text{Expr} \) can be unified
  - \( X \) may or may not be instantiated
  - \( \text{Expr} \) must not contain any uninstantiated variables
  - As a side effect, \( X \) is instantiated to the arithmetic evaluation of \( \text{Expr} \)
  - E.g.
    - `\|\| X \equiv (3 \times 7) + 1 \equiv 4 Yes`
    - `\|\| X \equiv (3 \times 4) + 10 \equiv \text{mod} 6 X=4`
    - `\|\| \text{is}(2 + 3, 5). X=4 No`
    - `\|\| \text{is}(2 + 3, 5). X=4 Yes`

### Prolog: lists & unification

- **Examples**:
  - `[A, B, C] = [a, b, c, d, e, f], A = a B = b C = [c, d, e, f]`
  - `[A, B, C] = [a, b, c, d, e, f], A = a B = b C = [c, d, e, f]`
  - `[X, Y] = [john, skates], X=john Y=skates
  - `[cat] = [H|T], H = cat T = []`
  - `[(the,Y)]Z = [(X|hare)|is|here]], Y = hare Z = [[is, here]], X = the
  - `[H|T] = a(b, c(d)), Error`
  - `in(X,Y); a(1)) = [Name,Age] X = _G17 Y = _G18 Name = in(_G17, _G18) Age = a(1)"
**Prolog: recursion**

- **Recursively defined predicate:** if a predicate symbol occurs both in the head and body of a rule, then the rule is recursive.
  - E.g. \( a(X) \rightarrow b(X,Y), a(Y) \).
  
- **Mutually recursive predicates:** recursion might be indirect, involving several rules.
  - E.g. \( a(X) \rightarrow b(X,Y), c(Y) \).
    \( c(Y) \rightarrow d(Y,Z), a(Z) \).
    The predicates \( a \) and \( c \) are said to be mutually recursive.

- **Non-linear recursion:**
  - E.g. \( a(X) \rightarrow b(X,Y), a(Y), c(Y,Z), a(Z) \).
  
**Prolog: recursion – examples**

- **Factorial:**
  - Declarative Semantics:
    - Factorial is 1 if \( n = 0 \), else Factorial is \( n \times \text{factorial}(n-1) \)
  - Prolog:
    
    \[
    \begin{align*}
    &\text{factorial}(0,1). \\
    &\text{factorial}(Y,X) : \text{if } Y > 0, \text{ then } \text{factorial}(Y-1,X1), X \leftarrow Y \times X1.
    \end{align*}
    \]

- **Appending lists:**
  - Declarative Semantics:
    - Appending an empty list to a non-empty list is the non-empty list.
    - Otherwise, work on one list by removing its elements and appending it to the other list.
  - Prolog:
    
    \[
    \begin{align*}
    &\text{append}([],X,X). \\
    &\text{append}([H | X], Y, [H | Z]) \leftarrow \text{append}(X,Y,Z).
    \end{align*}
    \]

- **Member of a list:**
  - Declarative Semantics:
    - \( X \) is a member of a list if \( X \) is equal to the first element, or a member of any sublist of that list
  - Prolog:
    
    \[
    \begin{align*}
    &\text{member}(X,[X|T]). \\
    &\text{member}(X,[Y|T]) \leftarrow \text{member}(X,T).
    \end{align*}
    \]

- **Blocks:**
  - Declarative Semantics:
    - Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:
    
    \[
    \begin{align*}
    &\text{above}(X,Y) \leftarrow \text{on}(X,Y). \quad \text{(1)} \\
    &\text{above}(X,Y) \leftarrow \text{above}(X,Z), \text{above}(Y,Z). \quad \text{(2)} \\
    &\text{on}(a,b). \quad \text{(3)} \\
    &\text{on}(b,c). \quad \text{(4)} \\
    &\text{on}(c,d). \quad \text{(5)} \\
    &? \text{above}(a,b). \\
    &? \text{above}(b,c). \\
    &? \text{above}(a,d). \\
    &? \text{above}(b,d). \\
    &? \text{above}(a,b). \\
    \end{align*}
    \]

Infinite recursion! Trace it to see why.

\[
\begin{align*}
? \text{above}(a,c). \\
\end{align*}
\]

Infinite recursion! Trace it to see why.
Prolog: recursion – examples

- Blocks:
  - Declarative Semantics:
    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

```
% Second attempt
above(X,Y) :- on(X,Y).                       (1)
above(X,Z) :- on(X,Y), above(Y,Z).  (2)
on(a,b).       (3)
on(b,c).        (4)
on(c,d).        (5)
%?- above(a,d).
Yes
```

- Note that sometimes changing the order of rules and/or rule premises can cause problems for Prolog

- Example: above(X,Z) :- on(X,Y), above(Y,Z).  (1) above(X,Y) :- on(X,Y).                       (2) on(a,b).       (3) on(b,c).        (4) on(c,d).        (5) %?- above(a,d).
Yes

Prolog: recursion – examples

- Infinite recursion:
  - E.g.: p :- p.
    - Declaratively perfectly correct, but procedurally causes infinite loop

- What to do about infinite recursion?
  - Rewrite the rules and facts (most widely used technique)
  - Define a second non-recursive version (similar to a base case)
  - Use ! to stop the unification (more about this later).

Prolog: complex types - structures

- Recall: what's a function term? 
  \[ \text{functor}(\text{some-parameters}) \]
  - E.g. woman(marry)

- We can construct complex data structures using nested function terms and lists.
  - Represents a statement about the world

- Example 2:
  - A family consist of 2 persons, and 0 or more children. Each person is either employed for some salary or unemployed.

```
family(person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),
person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),
[person(First-name,Last-name,date(Day,Month,Year),unemployed),
person(First-name,Last-name,date(Day,Month,Year),unemployed)])
```
Prolog: structures – example 2

• Family database:

\[
\begin{align*}
\text{family(} & \text{person(tom,fox,date(7, may, 1950), works(cbc, 15200)),} \\
& \text{person(ann,fox,date(9, may, 1951), works(ctv, 25700)),} \\
& \text{person(pat,fox,date(5, may, 1973), unemployed),} \\
& \text{person(jim,fox,date(5, may, 1973), unemployed))}.
\end{align*}
\]

To find if there is a married woman that 3% have at least three children:

\[
\begin{align*}
| ?- & \text{family(_,person(Name,Surname,_,_),[_,_,_|_]).} \\
\text{no} \\
| ?- & \text{family(} \text{person(tom,fox,date(7, may, 1950), works(cbc, 15200)),} \\
& \text{person(ann,fox,date(9, may, 1951), works(ctv, 25700)),} \\
& \text{[person(pat,fox,date(5, may, 1973), unemployed),} \\
& \text{person(jim,fox,date(5, may, 1973), unemployed))}. \\
\text{no} \\
\end{align*}
\]

To find if there is a family of three children:

\[
| ?- & \text{family(_,_,[_,_,_]).} \\
\text{no}
\]

To find all the fox families:

\[
| ?- & \text{family(person(X,fox,Y,Z),T,W).} \\
X = \text{tom} \\
Y = \text{date(7, may, 1950)} \\
Z = \text{works(cbc, 15200)} \\
T = \text{person(ann,fox,date(9, may, 1951), works(ctv, 25700))} \\
W = \text{[person(pat,fox,date(5, may, 1973), unemployed),} \\
& \text{person(jim,fox,date(5, may, 1973), unemployed))}. \\
\text{no}
\]

Let us add more useful rules:

\[
\begin{align*}
\text{husband(X)} : & - \text{family(X,_,_,T,W).} \\
\text{wife(X)} : & - \text{family(_,X,_,T,W).} \\
\text{child(X)} : & - \text{family(_,_,Children), member(X,Children).} \\
\text{member(X, [Y|L]) : } & - \text{member(X,Y).} \\
\text{exists(Person) : } & - \text{husband(Person); wife(Person).} \\
\text{salary(Person, Salary)} : & - \text{exists(person(Name, Surname, date(_,_,Year), works(_, Salary)), Year < 1975).} \\
\text{dateofbirth(Person, Date)} : & - \text{exists(person(_,_,_,_,_), Year < 1975).} \\
\text{exists(Person) : } & - \text{husband(Person); wife(Person); child(Person).} \\
\end{align*}
\]

Find the names of all the people in database:

\[
| ?- & \text{exists(person(Name, Surname, _, _)).} \\
\text{Name = tom} \\
\text{Surname = fox} \\
\text{Name = ann} \\
\text{Surname = fox} \\
\text{Name = pat} \\
\text{Surname = fox} \\
\text{Name = jim} \\
\text{Surname = fox} \\
\text{no}
\]

Find all children born in 1973:

\[
| ?- & \text{child(X), dateofbirth(X, date(_, _, 1973)).} \\
\text{X = person(pat,fox,date(5, may, 1973), unemployed),} \\
& \text{person(jim,fox,date(5, may, 1973), unemployed).} \\
\text{no}
\]

Let us add a rule to add the salaries:

\[
| ?- & \text{family(Husband,Wife,Children), total([Husband,Wife|Children], Income).} \\
\text{Husband = person(tom,fox,date(7, may, 1950), works(cbc, 15200))} \\
\text{Wife = person(ann,fox,date(9, may, 1951), works(ctv, 25700))} \\
\text{Children = [person(pat,fox,date(5, may, 1973), unemployed),} \\
& \text{person(jim,fox,date(5, may, 1973), unemployed))}. \\
\text{Income = 40000;} \\
\text{no}
\]

To retrieve the nth child of a family, we need to define how to get the nth element of a list:

\[
| ?- & \text{nthchild(N, Family, Child)} : - children(Family, ChildList), nth_member(N, ChildList, Child).} \\
\text{nth_member(N, [Y|L], X) : - nth_member(N, L), X = Y.} \\
\text{nth_member(N, [], X) : - N > 0; X = X.} \\
\text{exists(Person) : } - \text{children(Person, ChildList).} \\
\]

Now, let us define how to get the nth child. Note: I left children clause for you to define:

\[
| ?- & \text{nthchild(N, Family, Child)} : - children(Family, ChildList), nth_member(N, ChildList, Child).} \\
\text{nth_member(N, [Y|L], X) : - nth_member(N, L), X = Y.} \\
\text{nth_member(N, [], X) : - N > 0; X = X.} \\
\text{exists(Person) : } - \text{children(Person, ChildList).}
\]
Prolog: cut!

- Cut is a special mechanism that can be used to tell Prolog which previous choices it need not consider again when it backtracks.

- The cut is written `!` and is inserted between goals as a pseudo-goal, for example:

  ```prolog
  head(X) :- goal1a, goal1b, !, goal1c, goal1d. % rule 1
  head(X) :- goal2a, goal2b, goal2c. % rule 2
  head(X) :- goal3a, goal3b, goal3c, goal3d. % rule 3
  ```

- Is the location of the `!`, in the same clause, significant?
  - Yes, consider above example, placing `!` after `goal1b` means that only satisfying `goal1a` and `goal1b` is sufficient for your solution and you do not need to unify on any of the goals in rules 2&3.
  - Note that this does not mean that rule 1 will succeed, because that is dependent on `goal1c` and `goal1d` being true as well.

Prolog: cut! - examples

- Double-step function:

  ```prolog
  if X < 3 then Y = 0
  if 3 <= X and X < 6 then Y = 2
  if 6 <= X then Y = 4
  ```

  `% In Prolog
  f(X,0) :- X < 3. % rule 1
  f(X,2) :- 3 =< X, X < 6. % rule 2
  f(X,4) :- 6 =< X. % rule 3
  ```

  ```prolog
  | ?- f(4,Y). % query
  Y=4
  | ?- f(1,Y),2<Y. % query
  no
  ```

- Double-step function - cont’d:

  ```prolog
  if X < 3 then Y = 0
  if 3 <= X and X < 6 then Y = 2
  if 6 <= X then Y = 4
  ```

  `% In Prolog
  f(X,0) :- X < 3,!. % rule 1
  f(X,2) :- 3 =< X, X < 6,!. % rule 2
  f(X,4) :- 6 =< X. % rule 3
  ```

  ```prolog
  | ?- f(1,Y). % query
  no
  ```

-Prolog: cut! - when to use?

- Common uses of the `!`:
  - Tell the Prolog system that it has found the right rule for a particular goal.
    * If you get this far, you have picked the correct rule for this goal.
  - Tell the Prolog system to fail a particular goal immediately without trying for alternate solutions.
    * If you get to here, you should stop trying to satisfy the goal.
  - Terminate the generation of alternative solutions.
    * If you get to here, you have found the only solution to this problem, no point in looking for alternatives.

-Prolog: cut! - example

- Double-step function - cont’d:

  ```prolog
  if X < 3 then Y = 0
  if 3 <= X and X < 6 then Y = 2
  if 6 <= X then Y = 4
  ```

  What do we know about this function that Prolog doesn’t?

  `% same relations with `!
  f(X,0) :- X < 3. % rule 1
  f(X,2) :- 3 =< X, X < 6. % rule 2
  f(X,4) :- 6 =< X. % rule 3
  ```

  ```prolog
  Y=2
  ```

  `no`

In this example, we changed the procedural meaning of the program, but not the declarative meaning.
Prolog: cut! – example

- Double-step cont’d:
  % same relations with !
  \( f(X,0) \) :- \( X < 3 \), !. %rule1
  \( f(X,2) \) :- \( 3 =< X, X < 6 \), !. %rule2
  \( f(X,4) \) :- \( 6 =< X \). %rule3
  \| ?- f(5,Y). %query
  Y = 2

- Can we come up with a more efficient version?
  \( f(X,0) \) :- \( X < 3 \). %rule1
  \( f(X,2) \) :- \( X < 6 \). %rule2
  \( f(X,4) \) :- %rule3
  \| ?- f(5,Y). %query
  Y = 2

% What if we removed the cuts?
\( f(X,0) \) :- \( X < 3 \). %rule1
\( f(X,2) \) :- \( X < 6 \). %rule2
\( f(X,4) \) :- %rule3
\| ?- f(1,Y). %query
Y = 0; % right answer
Y = 2; % wrong answer, why?
Y = 4; % wrong answer, why?

Here, we changed the procedural and also the declarative meaning