Principles of Programming Languages
Lecture 5

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

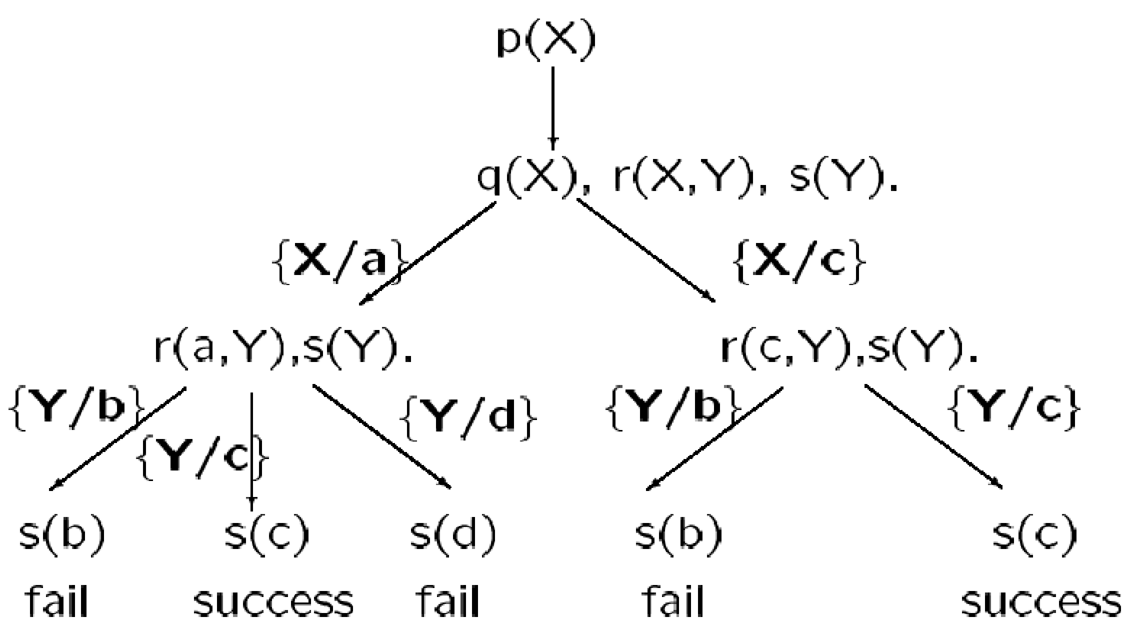
References: Scheme by Dybvig
Concepts of PL by Sebesta
Prog. in Prolog by Clocksin and Mellish
PL Concepts and Constructs by Sethi
ML for the Working Prog. By Paulson
PL Pragmatics by Scott

Prolog: backtracking example 2

Rule base:

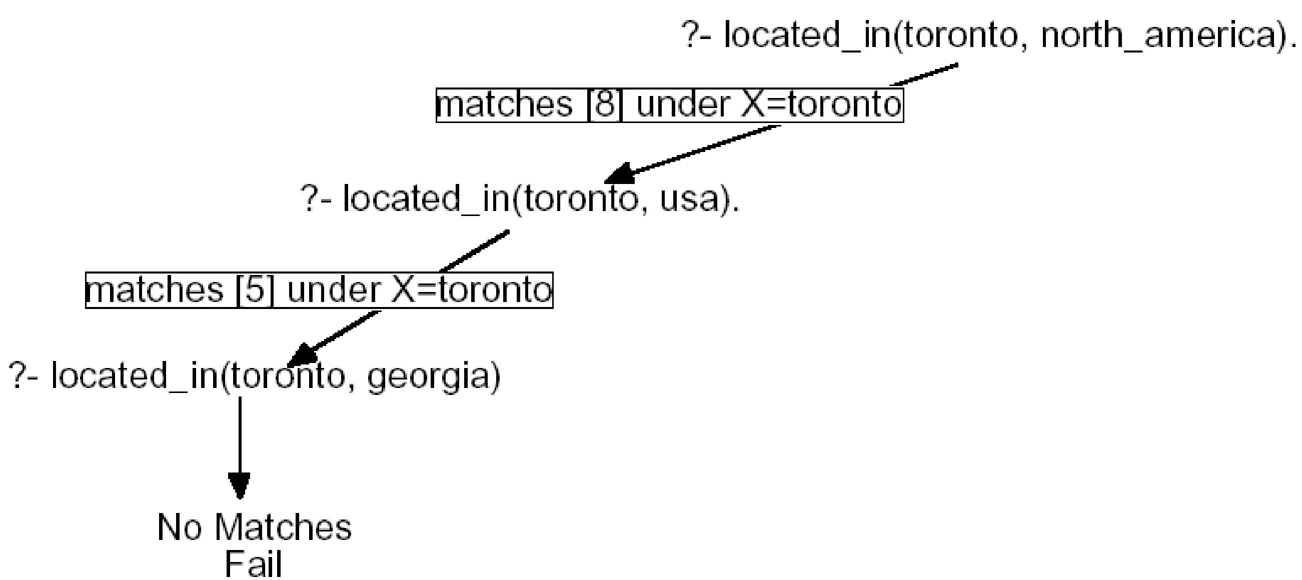
```
p(X) :- q(X),r(X,Y),s(Y).
q(a).     r(a,b).     r(c,b).     s(c).
q(c).     r(a,c).     r(c,c).
          r(a,d).
```

Query: Find `X` such that `p(X)` is true.



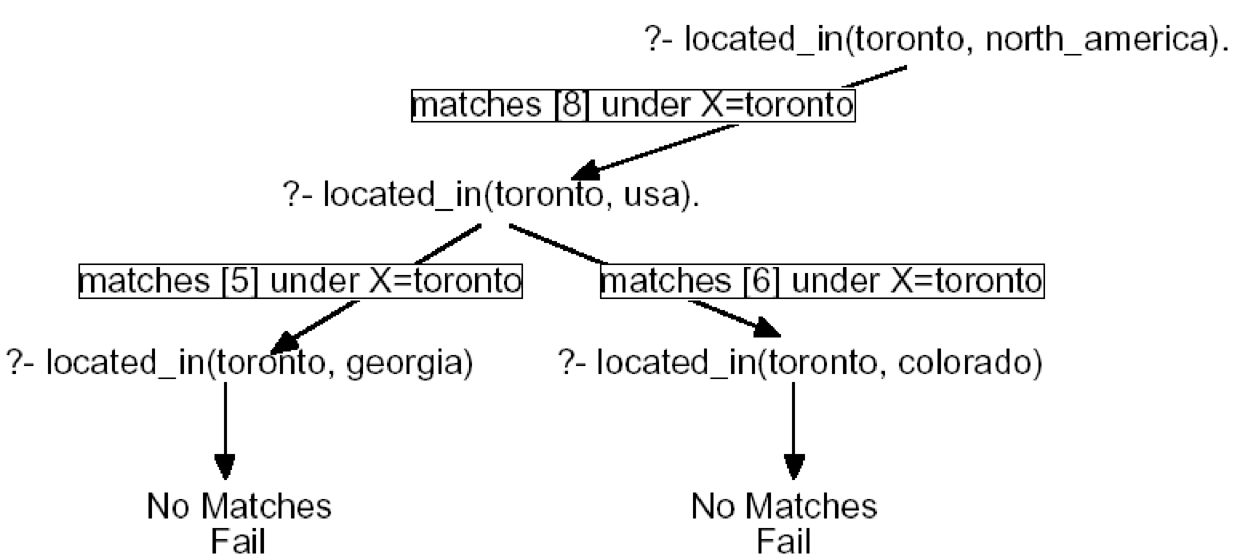
Prolog: backtracking example 3

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



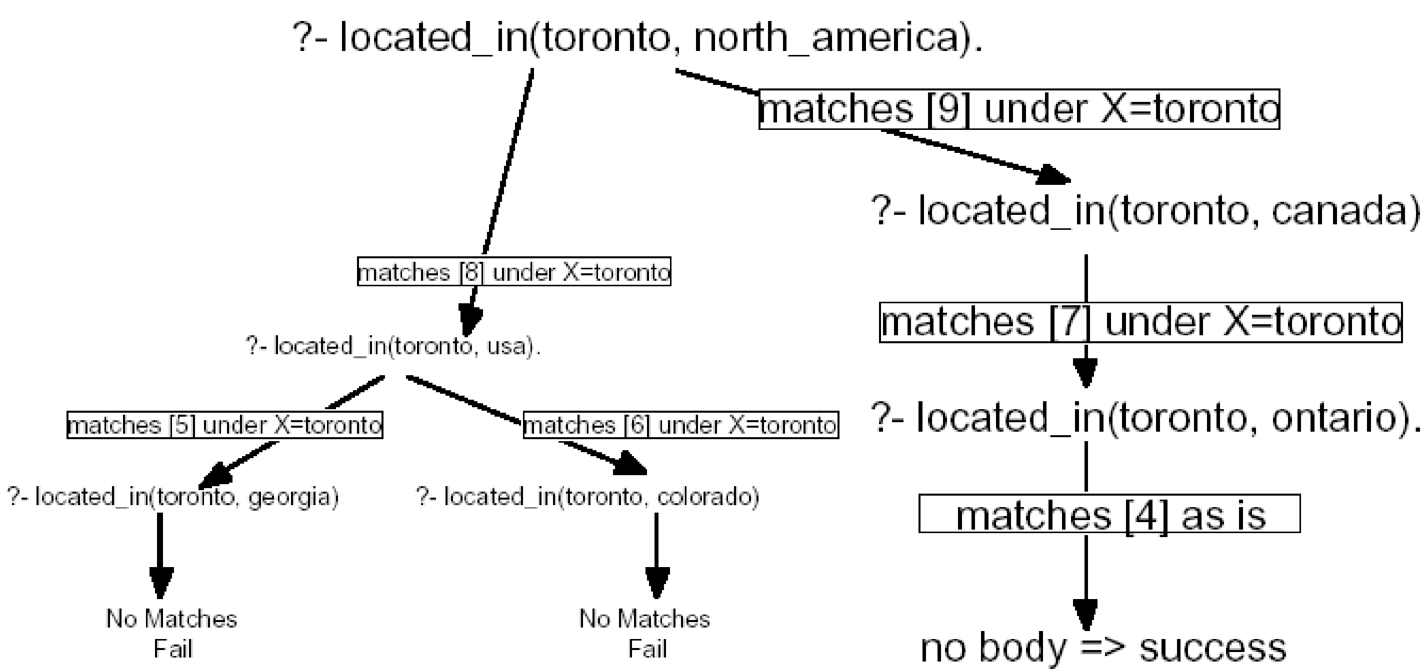
Prolog: backtracking example 3 – cont'd

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



Prolog: backtracking example 3 – cont’d

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



Prolog: top-down vs. bottom-up reasoning

- **Prolog uses top-down inference, although some other logic programming systems use bottom-up inference (e.g. Coral)**

- **Each has its own advantages and disadvantages:**
  - Bottom-up may generate many irrelevant facts
  - Top-down may explore many lines of reasoning that fail.

- **Top-down and bottom-up inference are logically equivalent**
  - i.e. they both prove the same set of facts.

- **However, only top-down inference simulates program execution**
  - i.e. execution is inherently top down, since it proceeds from the main procedure downwards, to subroutines, to sub-subroutines, etc...

Prolog: unification operators

- = *unify with* **operator:** **X = Y**
  - Semantically: unifable 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.

```
|? a(b,M,c)=a(b,10,d).
false

|? a(b,M,c)=a(b,10,c).
M = 10.

|? a(b(X))=a(b(Y)).
X = Y.
```

Prolog: unification operators

- \= ***does not unify with* operator**: X \= Y
  - Semantically: not-unifiable test
  - Succeeds as long as X and Y cannot be unified. X and Y must be instantiated.
  - E.g.

|? joe \= fred.
true

|? a(b,X,c) \= a(b,Y,c).
false

Prolog: unification operators

- == ***is already instantiated to*** operator: **X == Y**
  - Semantically: identical test
  - Succeeds as long as X and Y are already instantiated to the same object
  - No side effects
  - E.g.

    |? 4 == 2 +2 false

    |? a(b,X,c) == a(b,Y,c). false

    |? a(b,X,c) == a(b,X,c). true

- =:= ***is already instantiated to*** operator**:** **X =:= Y**
  - Semantically: identical test after evaluating terms
  - E.g.

    |? 4 =:= 2 + 2 true

    |? a(b,X,c) =:= a(b,Y,c). Error, a cannot be evaluated

Prolog: unification operators

- \== ***not already instantiated to*** operator**: X \== Y**
  - 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. true

    |? a(b,X,c) \== a(b,Y,c). true

    |? 1 +2 \== 3 true

- =\= ***is already instantiated to*** operator**: X =:= Y**
  - Semantically: not-identical test <u>after evaluating terms</u>
  - E.g.

    |? 4 =\= 2 + 2 false

Prolog: operators

- is operator: **X is Expr** **is(X,Expr)**
  - Semantically: 1) evaluate second term and 2) test if it is equal to X
  - succeeds a long as X and the arithmetic evaluation of Expr can be unified
  - X may or may not be instantiated
  - Expr must not contain any uninstantiated variables
  - As a side effect, X is instantiated to the arithmetic evaluation of Expr
  - E.g.

```
|? 5 is ( ( 3 * 7 ) + 1 ) //  4
true

|? X is ( ( 3 * 4 ) +10) mod 6
X=4

|? is(2+3,5).
false

|? is(5,2+3).
true
```

Prolog: lists

- **A sequence of terms of the form**

  [t_1, t_2, t_3, t_4, ..., t_n] *where term t_i is the ith 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 nested lists

- **Can break apart lists using "|" into [Head | 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: lists & unification

- **Examples:**
  - [X,Y] = [john, skates]. X=john Y=skates
  - [cat] = [H|T]. H = cat T = []
  - [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]
  - [[the,Y]|Z] = [[X,hare],[is,here]]. Y = hare Z = [[is, here]] X = the
  - [H|T] = a(b, c(d)). Error
  - [n(X,Y),a(1)] = [Name,Age]. Name = n(X, Y) Age = a(1)

Prolog: recursion

- **Recursively defined predicate:** *if a predicate symbol occurs both in in the head and and body of a rule, then the rule is recursive.*
  - E.g. a(X) :- b(X,Y), a(Y).

    This predicate acts like a recursive subroutine.

- **Mutually recursive predicates:** *recursion might be indirect, involving several rules.*
  - E.g. a(X) :- b(X,Y), c(Y).

    c(Y) :- d(Y,Z), a(Z).

    The predicates a and c are said to be mutually recursive.

- **Non-linear recursion:**
  - E.g. a(X) :- b(X,Y), a(Y), c(Y,Z), a(Z).

    This generates what we call a recursive proof tree.

Prolog: recursion

- **Recall: how to code recursion?**

1. Identify Base case ➔ **a rule without body. Comes first.**

2. Identify recursive case ➔ **recursive rule.**

Prolog: recursion – examples

- **Factorial:**

  $$n! \equiv n(n-1)\cdots 2\cdot 1.$$

  - Declarative Semantics:

    Factorial is 1 if n = 0, else Factorial is n * factorial (n-1)

  - Java

```
public long factorial( int n ) {
        if( n <= 1 )    // base case
                return 1;
        else
                return n * factorial( n - 1 );
}
```

factorial(4) ➔ 4 * factorial (3)

3 * factorial (2)

2 * factorial (1)

1

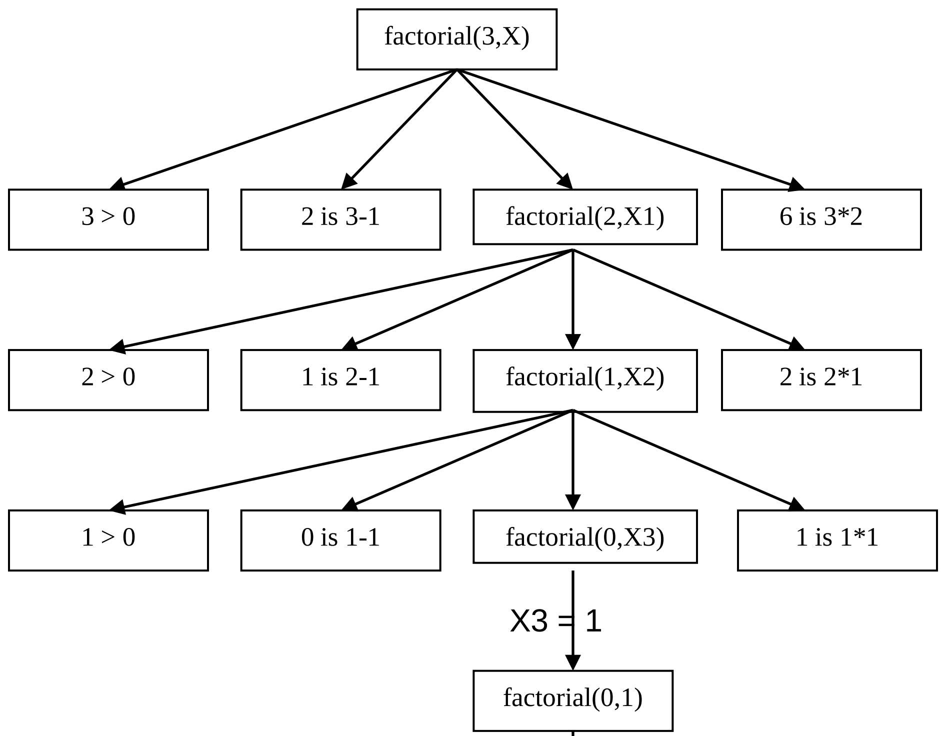
factorial(4) ➔ 4 * 3 * 2 * 1

Prolog: recursion – examples

- **Factorial:** $n! = n(n-1)\cdots 2\cdot 1.$
  - Declarative Semantics:

    Factorial is 1 if n = 0, else Factorial is n * factorial (n-1)
  - Prolog:

```
factorial(0,1).
factorial(Y,X) :- Y>0,  Y1 is Y-1, factorial(Y1,X1), X is Y*X1.
```



Prolog: recursion – examples

- **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:

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