Programming Paradigms CSI2120 – Winter 2018

Jochen Lang
EECS, University of Ottawa
Canada

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

Logic Programming in Prolog

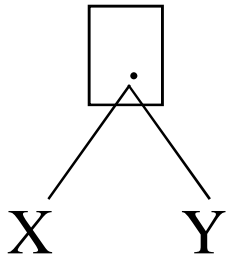
- **List Representations**
  - dot operator
  - Trees and Vine diagrams
- **More List Processing**
  - Insertion vs. Deletion
  - List processing
    - Double Recursion
    - Accumulators
  - Examples

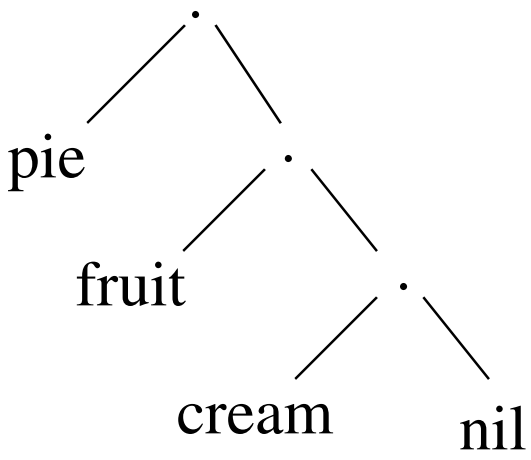
List Representations

- **Lists can be written with the binary function symbol .**
  - Instead of using the familar $\{e_1, e_2, \ldots\}$ using the binary function the list is $(e_1.(e_2.(\ldots)))$
- **The empty list is also noted as nil and is the end marker of the list**
- **Examples :**
  - The list {pie, fruit, cream} is written (pie.(fruit.(cream.nil)))
  - The variables X followed by Y can be written as (X.Y)

Tree Representation of Lists

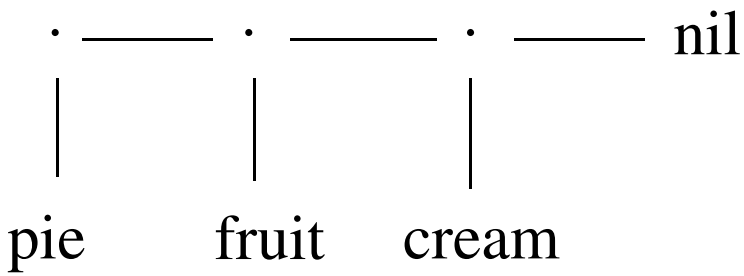
- **Examples:**





Fundamental List Properties

- **A list is represented by a particular tree with all the leaves of the tree on branches to the left.**
  - Sometimes the tree is shown in a vine diagram



- **Example :**
  - Solve the equation X.Y = pie.fruit.cream.nil
  - Solution:
    - {X = pie; Y = fruit.cream.nil}

Fundamental List Properties (cont'd)

- **The notation X.Y represents the head X and the tail Y of the list.**
- **The head is the first element and Y is the tail of the list.**
  - The notion of head and tail of a list is the basis of list processing in Prolog.
  - But the term X.Y is not a list just a pair (need brackets for a list)

Variable Number of Arguments

- **Consider a predicate province that associates cities with provinces, e.g.,**

```
province(ontario,toronto,ottawa,hamilton,kitchener,
         london).
province(quebec,montreal,quebec_city,sherbrooke,
         trois_rivieres).
province(new_brunswick,st_john,moncton,fredericton).
```

  - This will not work because the number of arguments in the predicate province is variable
- **Instead use a list and the predicate `province/2`**

```
province(quebec, montreal.( quebec_city .(sherbrooke
         .(trois_rivieres.nil)))).
province(new_brunswick, st_john .(moncton
         .(fredericton.nil))).
province(ontario, toronto .(ottawa .(hamilton
         .(kitchener .(london.nil))))).
```

Access to the Elements of a List

- **In Prolog**
  (montreal .(quebec_city .(sherbrooke .(trois_rivieres. nil))))
  using the symbol [ ] becomes
  [montreal, quebec_city, sherbrooke, trois_rivieres]
- **Accessing the elements of a list, we use [head|tail].**
- **The following are all the same list**
  [st_john, moncton, fredericton]
  [st_john | [moncton,fredericton]]
  [st_john | [moncton | [fredericton]]]
  [st_john | [moncton | [fredericton | [ ]]]]
  [st_john, moncton | [fredericton]]
  [st_john, moncton, fredericton | []]

Use of Member Predicate

- **Rule `inProvince/2` that returns the city's province**

```
inProvince(X,P) :- province(P, L),
                   member(X, L).
```

- **Queries**

```
?- inProvince(ottawa, P).
P = ontario ;
false.

?- inProvince(C, new_brunswick).
C = st_john ;
C = moncton ;
C = fredericton ;
false.
```

List insertion and deletion

- **Consider the list insertion predicate from before**

```
listInsert(A,L,[A|L]).
listInsert(A,[X|L], [X|LL]) :-
    listInsert(A,L,LL).
```

  - Query with the list after insertion and get the list before.

```
?- listInsert(a,L,[b,a,d,a,f]).
L = [b, d, a, f] ;
L = [b, a, d, f] ;
false.
```

  - As a generator, it can produce different solutions because the element `a` can be removed from different positions

Delete Elements from a List

- **Deletion of the first occurrence of an element**

```
deleteFront(R,[R|L],L).    % Element found
deleteFront(R,[X|LL],[X|L]) :-
      deleteFront(R,LL,L). % Use Accumulator
```

- **Delete all occurrences of an element**

```
deleteAll(_,[],[]).
deleteAll(X,[X|T],Result) :-
      deleteAll(X,T,Result),!.  % Delete once
deleteAll(X,[H|T],[H|Result]) :-
      deleteAll(X,T,Result).    % Other element
```

Intersection of two lists (set operations)

Lists can represent sets in Prolog

- **Simplified intersection assuming the input lists contain no duplicate elements, i.e., they are sets themselves**

```
intersectList( [], _, [] ).
intersectList( [ X | Xs ], Ys, Zs ) :-
                \+member( X, Ys),  % built-in
                intersectList( Xs, Ys, Zs ).
intersectList( [ X | Xs ], Ys, [ X | Zs ] ) :-
                member( X, Ys ),
                intersectList( Xs, Ys, Zs ).
```

- **Note there is also a library predicate `intersection/3`**

Quick Sorting a List

- **Recursive sorting**
- **Quicksort with simply selecting the first element as the pivot, better to randomize**

```
sortList([],[]).
sortList([P|Q],T) :- partitionList(P,Q,G,D)
          sortList(G,GG),
          sortList(D,DD),
          append(GG,[P|DD],T).
```

  - Making use of the library predicate append (could use our own definition appendList from before).
  - Needs partitionList predicate (next slide).

Partioning a List

- **Splitting a list into 2 lists with a pivot**
  - One list greater than the pivot
  - One list smaller than the pivot
  - Use alphanumeric comparison operator
    - Instead could use an additional rule `lessThan(X,P)`

```
partitionList(P,[X|L],[X|PG],PD) :- X @< P,
        partitionList(P,L,PG,PD).
partitionList(P,[X|L],PG,[X|PD]) :- X @>= P,
        partitionList(P,L,PG,PD).
partitionList(P,[],[],[]).
```

Invert the Order of a List

```
mirror([],[]). % empty list is mirrored itself
mirror([X|L1], L2) :-    % Take X off the front
    mirror(L1,L3),       % Mirror the rest L1
    append(L3, [X], L2). % Put X at the back
```

- Note we use built-in append which behaves as appendList
- Queries

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

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

Improved List Inversion

```
reverseList([],L,L) :- !.
reverseList([H|T],L,R) :-
        reverseList(T,[H|L],R).

mirrorAcc(L,R) :- reverseList(L,[],R).
```

- **Note the use of the Cut in the boundary case**
- **Example**

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

Comparing List Inversion

- **First mirror predicate uses double recursion**
  - first recursion on mirror
  - second recursion with append
  - Note: replace `append/3` with `appendList/3` and trace
- **Second mirrorAcc predicate uses an accumulator**
  - Not instantiated variable is passed as an argument to the base case
  - Once reached it is unified all the way up the call stack
  - Improved efficiency compared with double recursion
    - Only one recursion. Call stack has a depth equals the length of the list.

Operators on a List

- **Example: Apply an operator to each element of a list**

```
applyToList([],[]).              % boundary case
applyToList([X|L],[Y|T]) :- applyOp(X,Y),
                            applyToList(L,T).
```

- **Example: Sum up the elements of a list of numbers**

```
sumList(L,S) :- sumList(L,0,S). % Helper
sumList([],S,S).  % Extra argument for result
sumList([X|L],T,S) :- TT is T+X,
                      sumList(L,TT,S).
```

Summary

- **List Representations**
  - dot operator
  - Trees and Vine diagrams
- **More List Processing**
  - Insertion vs. Deletion
  - List processing
    - Double Recursion
    - Accumulators
  - Examples