

*Logic Programming*  
*Using Data Structures*  
*Part 2*

Temur Kutsia

Research Institute for Symbolic Computation  
Johannes Kepler University Linz, Austria  
`kutsia@risc.jku.at`

# Contents

Recursive Comparison

Joining Structures Together

Accumulators

Difference Structures

# Comparing Structures

Structure comparison:

- ▶ More complicated than the simple integers
- ▶ Have to compare all the individual components
- ▶ Break down components recursively.

# Comparing Structures. `alless`

## Example

`alless(X, Y)` succeeds if

- ▶ `X` and `Y` stand for atoms and
- ▶ `X` is alphabetically less than `Y`.

`alless(avocado, clergyman)` **succeeds**.

`alless(windmill, motorcar)` **fails**.

`alless(picture, picture)` **fails**.

## Comparing Structures. `aless`

Success Empty word is smaller than a nonempty one.

Success The first character of the first word is alphabetically less than one of the second:

```
aless(avocado, clergyman).
```

Recursion The first character is the same in both. Then have to check the rest:

```
For aless(lazy, leather) check  
aless(azy, eather).
```

Failure The first character of the first word is greater than the first one of the second:

```
aless(book, apple).
```

Failure Reach the end of both words at the same time:

```
aless(apple, apple).
```

Failure Run out of characters for the second word:

```
aless(alphabetic, alp).
```

# Representation

- ▶ Transform atoms into a recursive structure.
- ▶ List of integers (ASCII codes).
- ▶ Use built-in predicate `atom_codes`:

```
?- atom_codes(alp, [97,108,112]).  
yes
```

```
?- atom_codes(alp, X).  
X = [97,108,112] ?  
yes
```

```
?-atom_codes(X, [97,108,112]).  
X = alp ?  
yes
```

# First Task

Convert atoms to lists:

```
atom_codes(X, XL) .  
atom_codes(Y, YL) .
```

Compare the lists:

```
allessx(XL, YL) .
```

Putting together:

```
alless(X, Y) :-  
  atom_codes(X, XL),  
  atom_codes(Y, YL),  
  allessx(XL, YL) .
```

## Second Task

Compose `allessx`.

Success The first word ends before the second:

```
allessx([], [_|_]).
```

Success The first character in the first is alphabetically less than the the one in the second:

```
allessx([X|_], [Y|_]) :- X < Y.
```

Recursion The first character is the same in both. Then have to check the rest:

```
allessx([H|X], [H|Y]) :- allessx(X, Y).
```

What about failing cases?



# Program

```
alless(X, Y):-  
  atom_codes(X, XL),  
  atom_codes(Y, YL),  
  alessx(XL, YL).  
  
alessx([], [_|_]).  
alessx([X|_], [Y|_]):-  
  X < Y.  
alessx([H|X], [H|Y]):-  
  alessx(X, Y).
```

# Appending Two Lists

For any lists `List1`, `List2`, and `List3`  
`List2` **appended** to `List1` is `List3` iff either

- ▶ `List1` is the empty list and `List3` is `List2`, or
- ▶ `List1` is a nonempty list and
  - ▶ the head of `List3` is the head of `List1` and
  - ▶ the tail of `List3` is `List2` **appended** to the tail of `List1`.

Program:

```
append([], L, L).  
append([X|L1], L2, [X|L3]) :-  
    append(L1, L2, L3).
```

## Using append

**Test** ?- append([a,b,c],[2,1],[a,b,c,2,1]).

**Total List** ?- append([a,b,c],[2,1],X).

**Isolate** ?- append(X,[2,1],[a,b,c,2,1]).

?- append([a,b,c],X,[a,b,c,2,1]).

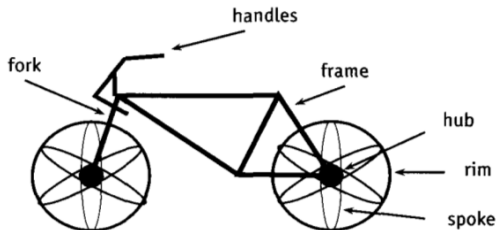
**Split** ?- append(X,Y,[a,b,c,2,1]).

# Inventory Example: Bicycle Factory

To build a bicycle we need to know which parts to draw from the supplies.

Each part of a bicycle may have subparts.

Task: Construct a tree-based database that will enable users to ask questions about which parts are required to build a part of bicycle.



# Parts of a Bicycle

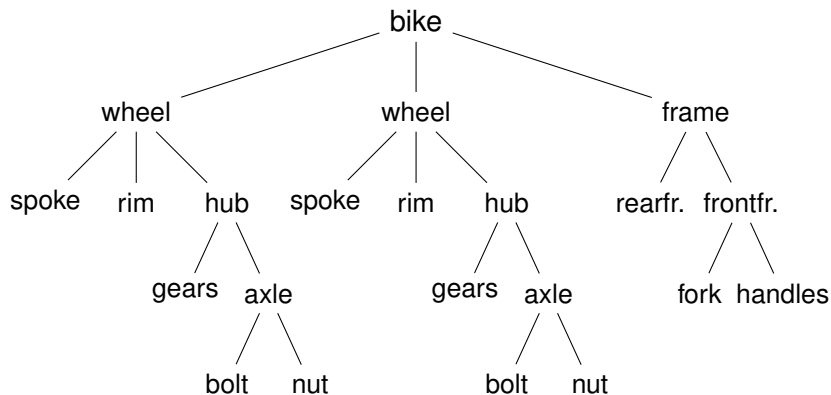
- Basic parts:

```
basicpart(rim).          basicpart(gears).  
basicpart(spoke).        basicpart(bolt).  
basicpart(rearframe).    basicpart(nut).  
basicpart(handles).      basicpart(fork).
```

- Assemblies, consisting of a quantity of basic parts or other assemblies:

```
assembly(bike, [wheel, wheel, frame]).  
assembly(wheel, [spoke, rim, hub]).  
assembly(frame, [rearframe, frontframe]).  
assembly(hub, [gears, axle]).  
assembly(axle, [bolt, nut]).  
assembly(frontframe, [fork, handles]).
```

# Bike as a Tree



# Program

Write a program that, given a part, will list all the basic parts required to construct it.

Idea:

1. If the part is a basic part then nothing more is required.
2. If the part is an assembly, apply the same process (of finding subparts) to each part of it.

## Predicates: `partsof`

`partsof(X, Y)`: Succeeds if `X` is a part of bike, and `Y` is the list of basic parts required to construct `X`.

- ▶ Boundary condition. Basic part:

```
partsof(X, [X]) :- basicpart(X).
```

- ▶ Assembly:

```
partsof(X, P) :-  
  assembly(X, Subparts),  
  partsoflist(Subparts, P).
```

- ▶ Need to define `partsoflist`.



## Predicates: `partsoflist`

- Boundary condition. List of parts for the empty list is empty:

```
partsoflist([], []).
```

- Recursive case. For a nonempty list, first find `partsof` of the head, then recursively call `partsoflist` on the tail of the list, and glue the obtained lists together:

```
partsoflist([P|Tail], Total) :-  
    partsof(P, Headparts),  
    partsoflist(Tail, Tailparts),  
    append(Headparts, Tailparts, Total).
```

► The same example using accumulators

## Finding Parts

```
?- partsof(bike, Parts).  
Parts=[spoke,rim,gears,bolt,nut,spoke,rim,  
gears,bolt,nut,rearframe,fork,handles] ;  
false.
```

```
?- partsof(wheel, Parts).  
Parts=[spoke, rim, gears, bolt, nut] ;  
false.
```

# Using Intermediate Results

Frequent situation:

- ▶ Traverse a PROLOG structure.
- ▶ Calculate the result which depends on what was found in the structure.
- ▶ At intermediate stages of the traversal there is an intermediate value for the result.

Common technique:

- ▶ Use an argument of the predicate to represent the “answer so far”.
- ▶ This argument is called an accumulator.

# Length of a List without Accumulators

## Example

`listlen(L, N)` succeeds if the length of list `L` is `N`.

- ▶ Boundary condition. The empty list has length 0:  
`listlen([], 0).`
- ▶ Recursive case. The length of a nonempty list is obtained by adding one to the length of the tail of the list.

```
listlen([H|T], N) :-  
    listlen(T, N1),  
    N is N1 + 1.
```

# Length of a List with an Accumulator

## Example

`lenacc(L, A, N)` succeeds if the length of list `L`, when added the number `A`, is `N`.

- ▶ Boundary condition. For the empty list, the length is whatever has been accumulated so far, i.e. `A`:  
`lenacc([], A, A).`
- ▶ Recursive case. For a nonempty list, add 1 to the accumulated amount given by `A`, and recur to the tail of the list with a new accumulator value `A1`:

```
lenacc([H|T], A, N) :-  
  A1 is A + 1,  
  lenacc(T, A1, N).
```

# Length of a List with an Accumulator, Cont.

## Example

Complete program:

```
listlenacc(L, N) :-  
  lenacc(L, 0, N).  
  
lenacc([], A, A).  
lenacc([H|T], A, N) :-  
  A1 is A + 1,  
  lenacc(T, A1, N).
```

# Computing List Length

## Example (Version without an accumulator)

```
listlen([a,b,c], N).  
listlen([b,c], N1), N is N1 + 1.  
listlen([c], N2), N1 is N2 + 1, N is N1 + 1.  
listlen([], N3), N2 is N3 + 1, N1 is N2 + 1,  
    N is N1 + 1.  
N2 is 0 + 1, N1 is N2 + 1, N is N1 + 1.  
N1 is 1 + 1, N is N1 + 1.  
N is 2 + 1.  
  
N = 3
```

# Computing List Length

## Example (Version with an accumulator)

```
listlenacc([a,b,c], N).  
lenacc([a,b,c], 0, N).  
A1 is 0+1, lenacc([b,c], A1, N).  
lenacc([b,c], 1, N).  
A2 is 1+1, lenacc([c], A2, N).  
lenacc([c], 2, N).  
A3 is 2+1, lenacc([], A3, N).  
lenacc([], 3, N).
```

$N = 3$



# List as an Accumulator

Accumulators need not be integers.

If a list is to be produced as a result, an accumulator will hold a list produced so far.

Wasteful joining of structures avoided.

## Example (Reversing Lists)

```
reverse(List, Rev) :-  
  rev_acc(List, [], Rev).
```

```
rev_acc([], Acc, Acc).  
rev_acc([X|T], Acc, Rev) :-  
  rev_acc(T, [X|Acc], Rev).
```

# Bicycle Factory

Recall how parts of bike were found. ► Inventory example

`partsof` list has to find the parts coming from the list  
`[wheel, wheel, frame]`:

- **Find** parts of `frame`.
- **Append** them to `[]` to find parts of `[frame]`.
- **Find** parts of `wheel`.
- **Append** them to the parts of `[frame]` to find parts of `[wheel, frame]`.
- **Find** parts of `wheel`.
- **Append** them to the parts of `[wheel, frame]` to find parts of `[wheel, wheel, frame]`.

Wasteful!

# Bicycle Factory

Improvement idea: Get rid of append. Use accumulators.

`partsacc(X, A, P) : parts of X, when added to A, give P.`

`partsof(X, P) :- partsacc(X, [], P).`

`partsacc(X, A, [X|A]) :- basicpart(X).`

`partsacc(X, A, P) :-`

`assembly(X, Subparts),`

`partsacclist(Subparts, A, P).`

`partsacclist([], A, A).`

`partsacclist([P|Tail], A, Total) :-`

`partsacc(P, A, Headparts),`

`partsacclist(Tail, Headparts, Total).`

# Difference Structures

Compute parts of wheel without and with accumulator:

## Example (Without Accumulator)

```
?- partsof(wheel, P).  
X = [spoke, rim, gears, bolt, nut] ;  
No
```

## Example (With Accumulator)

```
?- partsof(wheel, P).  
X = [nut, bolt, gears, rim, spoke] ;  
No
```

Reversed order.

# Difference Structures

How to avoid wasteful work and retain the original order at the same time?

Difference structures.

# Open Lists and Difference Lists

- ▶ Consider the list  $[a, b, c \mid H\circ]$ .
- ▶ The structure of the list is known up to a point.
- ▶ If, at some point,  $H\circ$  is unbound then we have an **open list**.
- ▶ Informally,  $H\circ$  is called a “hole”.

# Open Lists and Difference Lists

- Unify  $Ho$  with  $[d, e]$ :

`?- List=[a,b,c|Ho], Ho=[d,e].`

`List=[a,b,c,d,e]`

- We started with an open list and “filled” in the hole with the structure.

# Open Lists and Difference Lists

- ▶ The result of filling in the hole in an open list with a “proper” list is a “proper” list.
- ▶ What happens if we instantiate the hole with an open list?
- ▶ The result will be an open list again:

```
?- List=[a,b,c|Ho], Ho=[d,e|Y].
```

```
?- List=[a,b,c,d,e|Y].
```



# Open Lists and Difference Lists

- ▶ Filling in the hole with a proper list, again:
- ▶ `?- List=[a,b,c|Ho], Ho=[d,e].`
- ▶ `?- List=[a,b,c,d,e].`
- ▶ Is not it the same as `append([a,b,c],[d,e],List)?`

## open\_append

- ▶ We can define `append` in terms of “hole filling”.
- ▶ Assume the first list is given as an open list.
- ▶ Define a predicate that fills in the hole with the second list.
- ▶ A naive and limited way of doing this:

```
open_append([H1,H2,H3|Hole],L2):-Hole=L2.
```

```
?- List=[a,b,c|Ho], open_append(List,[d,e]).
```

```
    List=[a,b,c,d,e]
```

```
    Ho=[d,e]
```

- ▶ Improvement is needed: This version assumes having a list with three elements and the hole.

# Improvement Idea

- ▶ One often wants to say about open lists something like  
*“take the open list and fill in the hole with ...”*
- ▶ Hence, one should know **both** an open list and a hole.
- ▶ Idea for list representation: Represent a list as an open list **together** with the hole.
- ▶ Such a representation is called a **difference list**.
- ▶ Example: The difference list representation of the list  $[a, b, c]$  is the pair of terms  $[a, b, c | X]$  and  $X$ .

## diff\_append

- **Difference append:**

```
diff_append(OpenList, Hole, L2) :- Hole=L2.  
?- List=[a,b,c|Ho], diff_append(List,Ho,[d,e]).  
    List=[a,b,c,d,e]  
    Ho=[d,e]
```

- **Compare to the open\_append:**

```
open_append([H1,H2,H3|Hole], L2) :- Hole=L2.  
?- List=[a,b,c|Ho], open_append(List,[d,e]).  
    List=[a,b,c,d,e]  
    Ho=[d,e]
```

# Difference Lists

- ▶ Introduce a notation for difference lists.
- ▶ Idea: We are usually interested the open list part of difference list, without the hole.
- ▶ From the pair  $[a, b, c \mid H_0]$  and  $H_0$  we are interested in  $[a, b, c]$ .
- ▶ “Subtracting” the hole  $H_0$  from the open list  $[a, b, c \mid H_0]$ .
- ▶  $[a, b, c \mid H_0] - H_0$ .
- ▶ The  $-$  has no interpreted meaning. Instead one could define any operator to use there.

## diff\_append. Version 2

► `diff_append(OpenList-Hole, L2) :- Hole=L2.`

`?- DList=[a,b,c|Ho]-Ho,`  
`diff_append(DList,[d,e]).`

`DList=[a,b,c,d,e]-[d,e]`  
`Ho=[d,e]`

► Has to be improved again: We are not interested in the “filled hole” in the instantiation of `Ho` hanging around.

## diff\_append. Version 3

- Let `diff_append` return the open list part of the first argument:

```
diff_append(OpenList-Hole, L2, OpenList) :-  
    Hole=L2.
```

```
?- DList=[a,b,c|Ho]-Ho,  
    diff_append(Dlist, [d,e], Ans) .  
  
    Dlist=[a,b,c,d,e]-[d,e]  
    Ho=[d,e]  
    Ans=[a,b,c,d,e]
```

- It is better now. `Ans` looks as we would like to.
- Still, there is a room for improvement: The `diff_append`
  - takes a difference list as its first argument,
  - a proper list as its second argument, and
  - returns a proper list.
- Let's make it more uniform.

## diff\_append. Version 3

- Better, but not the final approximation: `diff_append` takes two difference lists and returns an open list:

```
diff_append(  
  OpenList1-Hole1, OpenList2-Hole2, OpenList1  
) :-  
  Hole1=OpenList2.  
  
?- Dlist=[a,b,c|Ho]-Ho,  
   diff_append(Dlist,[d,e|Ho1]-Ho1,Ans).  
  
   Dlist=[a,b,c,d,e|Ho1]-[d,e|Ho1]  
   Ho=[d,e|Ho1]  
   Ans=[a,b,c,d,e|Ho1]
```

- We have returned an open list but we want a difference list.
- The first list has gained the hole of the second list.
- All we need to ensure is that we return the hole of the second list.



## diff\_append. Version 3

- Return the hole of the second list as well:

```
diff_append(  
  OpenList1-Hole1,  
  OpenList2-Hole2,  
  OpenList1-Hole2  
) :-  
  Hole1=OpenList2.  
  
?- DList=[a,b,c|Ho]-Ho,  
   diff_append(DList,[d,e|Ho1]-Ho1,Ans) .  
  
   DList=[a,b,c,d,e|Ho1]-[d,e|Ho1]  
   Ho=[d,e|Ho1]  
   Ans=[a,b,c,d,e|Ho1]-Ho1
```

- We have returned an difference list.
- Now we can recover the proper list we want:

```
?- DList=[a,b,c|Ho]-Ho,  
   diff_append(DList,[d,e|Ho1]-Ho1,Ans-[]) .  
  
   Ans=[a,b,c,d,e]
```

## diff\_append. Version 4

diff\_append can be made more compact:

```
diff_append(  
  OpenList1-Hole1,  
  Hole1-Hole2,  
  OpenList1-Hole2  
) .
```

## diff\_append. Usage

- Add an element at the end of a list:

```
add_to_back(L-H, El, Ans) :-  
    diff_append(L-H, [El|H1]-H1, Ans-[]).  
?- add_to_back([a,b,c|H]-H, e, Ans).  
   H = [e]  
   Ans = [a,b,c,e]
```

# Difference Structures

Both accumulators and difference structures use two arguments to build the output structure.

Accumulators: the “result so far” and the “final result”.

Difference structures: the (current approximation of the) “final result” and the “hole in there where the further information can be put”.

# Bicycle Factory

Use holes.

```
partsof(X, P) :-  
  partshole(X, P-Hole),  
  Hole=[].
```

```
partshole(X, [X|Hole]-Hole) :-  
  basicpart(X).  
partshole(X, P-Hole) :-  
  assembly(X, Subparts),  
  partsholelist(Subparts, P-Hole).
```

```
partsholelist([], Hole-Hole).  
partsholelist([P|Tail], Total-Hole) :-  
  partshole(P, Total-Hole1),  
  partsholelist(Tail, Hole1-Hole).
```

## Bicycle Factory. Detailed View

```
partsof(X, P) :-  
    partshole(X, P-Hole),  
    Hole=[].
```

- ▶ `partshole(X, P-Hole)` builds the result in the second argument `P` and returns in `Hole` a variable.
- ▶ Since `partsof` calls `partshole` only once, it is necessary to terminate the difference list by instantiating `Hole` with `[]`. (Filling the hole.)
- ▶ Alternative definition of `partsof`:  

```
partsof(X, P) :- partshole(X, P-[]).
```

  
It ensures that the very last hole is filled with `[]` even before the list is constructed.

## Bicycle Factory. Detailed View

```
partshole(X, [X|Hole]-Hole) :-  
    basicpart(X) .
```

- ▶ It returns a difference list containing the object (basic part) in the first argument.
- ▶ The hole remains open for further instantiations.

## Bicycle Factory. Detailed View

```
partshole(X, P-Hole):-  
    assembly(X, Subparts),  
    partsholelist(Subparts, P-Hole).
```

- ▶ Finds the list of subparts.
- ▶ Delegates the traversal of the list to `partsholelist`.
- ▶ The difference list `P-Hole` is passed to `partsholelist`.



## Bicycle Factory. Detailed View

```
partsholelist([P|Tail], Total-Hole) :-  
    partshole(P, Total-Hole1),  
    partsholelist(Tail, Hole1-Hole).
```

- ▶ `partshole` starts building the `Total` list, partially filling it with the parts of `P`, and leaving a hole `Hole1` in it.
- ▶ `partsholelist` is called recursively on the `Tail`. It constructs the list `Hole1` partially, leaving a hole `Hole` in it.
- ▶ Since `Hole1` is shared between `partshole` and `partsholelist`, after getting instantiated in `partsholelist` it gets also instantiated in `partshole`.
- ▶ Therefore, at the end `Total` consists of the portion that `partshole` constructed, the portion of `Hole1` `partsholelist` constructed, and the hole `Hole`.