Logic Programming

Lists. Recursion. Backtracking. The Cut predicate

October 21, 2015

- Lists
- Recursion
- Accummulators
- Backtracking and the Cut predicate

Introducing lists

- Lists are a common data structure in symbolic computation.

Introducing lists

- Lists are a common data structure in symbolic computation.
- Lists contain elements that are ordered.

Introducing lists

- Lists are a common data structure in symbolic computation.
- Lists contain elements that are ordered.
- Elements of lists are terms (any type, including other lists).

Introducing lists

- Lists are a common data structure in symbolic computation.
- Lists contain elements that are ordered.
- Elements of lists are terms (any type, including other lists).
- Lists are the only data type in LISP

Introducing lists

- Lists are a common data structure in symbolic computation.
- Lists contain elements that are ordered.
- Elements of lists are terms (any type, including other lists).
- Lists are the only data type in LISP
- They are *a* data structure in Prolog.

Introducing lists

- Lists are a common data structure in symbolic computation.
- Lists contain elements that are ordered.
- Elements of lists are terms (any type, including other lists).
- Lists are the only data type in LISP
- They are *a* data structure in Prolog.
- Lists can represent practically *any* structure.

Lists (inductive domain)

- "Base case": [ ] – the empty list.

Lists (inductive domain)

- “Base case”: `[ ]` – the empty list.
- “General case” : `.(h, t)` – the nonempty list, where:

Lists (inductive domain)

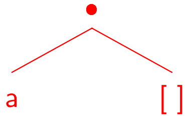
- "Base case": `[ ]` – the empty list.
- "General case" : `.(h, t)` – the nonempty list, where:
  - h - the head, can be any term,

Lists (inductive domain)

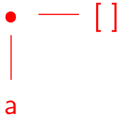
- "Base case": [ ] – the empty list.
- "General case" : .(h, t) – the nonempty list, where:
  - h - the head, can be any term,
  - t - the tail, must be a list.

List representations

- .(a, [ ]) is represented as



or

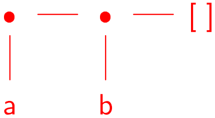


“tree representation”

“vine representation”

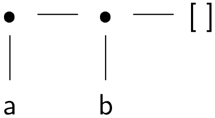
List representations (cont'd)

- .(a, .(b, [ ])) is

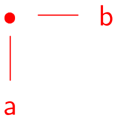


List representations (cont'd)

- .(a, .(b, [ ])) is

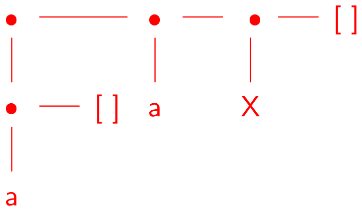


- .(a, b) is *not* a list, but it is a legal Prolog structure, represented as



List representations (cont'd)

- .(.(a , []), .(a , .(X, [ ]))) is represented as



Syntactic sugar for lists

- To simplify the notation, "," can be used to separate the elements.

Syntactic sugar for lists

- To simplify the notation, "," can be used to separate the elements.
- The lists introduced above are now:

Syntactic sugar for lists

- To simplify the notation, "," can be used to separate the elements.
- The lists introduced above are now:

  [a],

Syntactic sugar for lists

- To simplify the notation, "," can be used to separate the elements.
- The lists introduced above are now:

  [a],
  [a, b],

Syntactic sugar for lists

- To simplify the notation, "," can be used to separate the elements.
- The lists introduced above are now:

```
[a],
[a, b],
[[a], a, X].
```

List manipulation

- Lists are naturally split between the head and the tail.

List manipulation

- Lists are naturally split between the head and the tail.
- Prolog offers a construct to take advantage of this: [H | T].

List manipulation

- Lists are naturally split between the head and the tail.
- Prolog offers a construct to take advantage of this: [H | T].
- Consider the following example:

```
p([1, 2, 3]).
p([the, cat, sat, [on, the, mat]]).
```

List manipulation

- Lists are naturally split between the head and the tail.
- Prolog offers a construct to take advantage of this: [H | T].
- Consider the following example:

```
p([1, 2, 3]).
p([the, cat, sat, [on, the, mat]]).
```

- Prolog will give:

```
?-p([H | T]).
    H = 1,
    T = [2, 3];
    H = the
    T = [cat, sat, [on, the, mat]];
    no
```

List manipulation

- Lists are naturally split between the head and the tail.
- Prolog offers a construct to take advantage of this: [H | T].
- Consider the following example:

```
p([1, 2, 3]).
p([the, cat, sat, [on, the, mat]]).
```

- Prolog will give:

```
?-p([H | T]).
    H = 1,
    T = [2, 3];
    H = the
    T = [cat, sat, [on, the, mat]];
    no
```

- Attention! [a | b] is not a list, but it is a valid Prolog expression, corresponding to .(a, b)

Unifying lists: examples

```
[X, Y, Z] = [john, likes, fish]
 X = john
 Y = likes
 Z = fish
```

Unifying lists: examples

```
[X, Y, Z] = [john, likes, fish]
 X = john
 Y = likes
 Z = fish

 [cat] = [X | Y]
  X = cat
  Y = [ ]
```

Unifying lists: examples

```
[X, Y, Z] = [john, likes, fish]
 X = john
 Y = likes
 Z = fish

 [cat] = [X | Y]
  X = cat
  Y = [ ]

  [X, Y | Z] = [mary, likes, wine]
   X = mary
   Y = likes
   Z = [wine]
```

Unifying lists: examples (cont'd)

```
[[the, Y] | Z] = [[X, hare], [is, here]]
X = the
Y = hare
Z = [[is, here]]
```

Unifying lists: examples (cont'd)

```
[[the, Y] | Z] = [[X, hare], [is, here]]
X = the
Y = hare
Z = [[is, here]]

[golden | T] = [golden, norfolk]
T = [norfolk]
```

Unifying lists: examples (cont'd)

```
[[the, Y] | Z] = [[X, hare], [is, here]]
 X = the
 Y = hare
 Z = [[is, here]]

 [golden | T] = [golden, norfolk]
  T = [norfolk]

  [vale, horse] = [horse, X]
   false
```

Unifying lists: examples (cont'd)

```
[[the, Y] | Z] = [[X, hare], [is, here]]
 X = the
 Y = hare
 Z = [[is, here]]

 [golden | T] = [golden, norfolk]
  T = [norfolk]

  [vale, horse] = [horse, X]
   false

  [white|Q] = [P|horse]
   P = white
   Q = horse
```

Strings

- In Prolog, strings are written inside double quotation marks.
- Example: "a string".
- Internally, a string is a list of the corresponding ASCII codes for the characters in the string.
- ```
  ?- X = "a string".
     X = [97, 32, 115, 116, 114, 105, 110, 103]
  ```
  ```

Summary

- Items of interest:
  - the anatomy of a list in Prolog `.(h, t)`
  - graphic representations of lists: “tree representation”, “vine representation”,
  - syntactic sugar for lists `[...]`,
  - list manipulation: head-tail notation [H|T],
  - strings as lists,
  - unifying lists.

Induction/Recursion

- Inductive domain:

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest" (atomic) objects, that cannot be decomposed,

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest" (atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a “manageable way”, i.e.:
  - there are some “simplest”(atomic) objects, that cannot be decomposed,
  - there are “complex” objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the “simplest” objects.

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest"(atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest"(atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.
- Recursion is the dual of induction, i.e.:

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest" (atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.
- Recursion is the dual of induction, i.e.:
  - recursion describes computation in inductive domains,

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest"(atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.
- Recursion is the dual of induction, i.e.:
  - recursion describes computation in inductive domains,
  - recursive procedures (functions, predicates) call themselves,

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest" (atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.
- Recursion is the dual of induction, i.e.:
  - recursion describes computation in inductive domains,
  - recursive procedures (functions, predicates) call themselves,
  - but the recursive call has to be done on a "simpler" object.

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest"(atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.
- Recursion is the dual of induction, i.e.:
  - recursion describes computation in inductive domains,
  - recursive procedures (functions, predicates) call themselves,
  - but the recursive call has to be done on a "simpler" object.
  - As a result, a recursive procedure will have to describe the behaviour for:

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest" (atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.
- Recursion is the dual of induction, i.e.:
  - recursion describes computation in inductive domains,
  - recursive procedures (functions, predicates) call themselves,
  - but the recursive call has to be done on a "simpler" object.
  - As a result, a recursive procedure will have to describe the behaviour for:

  (a) The "simplest" objects, and/or the objects/situations for which the computation stops, i.e. the boundary conditions, and

Induction/Recursion

- Inductive domain:
  - A domain composed of objects constructed in a "manageable way", i.e.:
  - there are some "simplest"(atomic) objects, that cannot be decomposed,
  - there are "complex" objects that can be decomposed into **finitely** many simpler objects,
  - and this decomposition process can be performed finitely many times before one reaches the "simplest" objects.
  - In such domains, one can use **induction** as an inference rule.
- Recursion is the dual of induction, i.e.:
  - recursion describes computation in inductive domains,
  - recursive procedures (functions, predicates) call themselves,
  - but the recursive call has to be done on a "simpler" object.
  - As a result, a recursive procedure will have to describe the behaviour for:

  (a) The "simplest" objects, and/or the objects/situations for which the computation stops, i.e. the boundary conditions, and

  (b) the general case, which describes the recursive call.

Example: lists as an inductive domain

- simplest object: the empty list [ ].

Example: lists as an inductive domain

- simplest object: the empty list [ ].
- any other list is made of a head and a tail (the tail should be a list): [H|T].

Example: member

- Implement in Prolog the predicate member/2, such that member(X, Y) is true when X is a member of the list Y.

Example: member

- Implement in Prolog the predicate member/2, such that member(X, Y) is true when X is a member of the list Y.

```
% The boundary condition.
member(X, [X|_]).
% The recursive condition.
member(X, [_|Y]):-
            member(X, Y).
```

Example: member

- Implement in Prolog the predicate member/2, such that member(X, Y) is true when X is a member of the list Y.

```
% The boundary condition.
member(X, [X|_]).
% The recursive condition.
member(X, [_|Y]):-
            member(X, Y).
```

- The boundary condition is, in this case, the condition for which the computation stops (not necessarily for the "simplest" list, which is [ ]).

Example: member

- Implement in Prolog the predicate member/2, such that member(X, Y) is true when X is a member of the list Y.

```
% The boundary condition.
member(X, [X|_]).
% The recursive condition.
member(X, [_|Y]):-
            member(X, Y).
```

- The boundary condition is, in this case, the condition for which the computation stops (not necessarily for the "simplest" list, which is [ ]).
- For [ ] the predicate is false, therefore it will be omitted.

Example: member

- Implement in Prolog the predicate member/2, such that member(X, Y) is true when X is a member of the list Y.

```
% The boundary condition.
member(X, [X|_]).
% The recursive condition.
member(X, [_|Y]):-
            member(X, Y).
```

- The boundary condition is, in this case, the condition for which the computation stops (not necessarily for the "simplest" list, which is [ ]).
- For [ ] the predicate is false, therefore it will be omitted.
- Note that the recursive call is on a smaller list (second argument). The elements in the recursive call are getting smaller in such a way that eventually the computation will succeed, or reach the empty list and fail. predicate for the empty list (where it fails).

When to use the recursion?

- Avoid circular definitions:

```
parent(X, Y):- child(Y, X).
child(X, Y):- parent(Y, X).
```

When to use the recursion?

- Avoid circular definitions:

```
parent(X, Y):- child(Y, X).
child(X, Y):- parent(Y, X).
```

- Careful with left recursion:

```
person(X):-person(Y), mother(X, Y).
person(adam).
```

In this case,

```
?-person(X).
```

will loop (no chance to backtrack). Prolog tries to satisfy the rule and this leads to the loop.

- Order of facts, rules in the database:

```
is_list([A|B]):- is_list(B).
is_list([]).
```

The following query will loop:

```
?- is_list(X)
```

- The order in which the rules and facts are given matters. In general, **place facts before rules**.

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  “you are a computer” maps to “i am not a computer”,

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".

- Mapping procedure:

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".
- Mapping procedure:
  1. accept a sentence,

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".

- Mapping procedure:
  1. accept a sentence,
  2. change "you" to "i",

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".

- Mapping procedure:
  1. accept a sentence,
  2. change "you" to "i",
  3. change "are" to "am not",

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".

- Mapping procedure:
  1. accept a sentence,
  2. change "you" to "i",
  3. change "are" to "am not",
  4. change "french" to "german",

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".

- Mapping procedure:
  1. accept a sentence,
  2. change "you" to "i",
  3. change "are" to "am not",
  4. change "french" to "german",
  5. change "do" to "no",

Recursive mapping

- Mapping: given 2 similar structures, change the first into the second, according to some rules.
- Example:

  "you are a computer" maps to "i am not a computer",
  "do you speak french" maps to "i do not speak german".
- Mapping procedure:
  1. accept a sentence,
  2. change "you" to "i",
  3. change "are" to "am not",
  4. change "french" to "german",
  5. change "do" to "no",
  6. leave everything else unchanged.

Recursive mapping (continued)

- The program:

```
change(you, i).
change(are, [am, not]).
change(french, german).
change(do, no).
change(X, X).

alter([],[]).
alter([H|T], [X|Y]):-
        change(H, X),
        alter(T, Y).
```

Recursive mapping (continued)

- The program:

```
change(you, i).
change(are, [am, not]).
change(french, german).
change(do, no).
change(X, X).

alter([],[]).
alter([H|T], [X|Y]):-
        change(H, X),
        alter(T, Y).
```

- Note that this program is limited:

Recursive mapping (continued)

- The program:

```
change(you, i).
change(are, [am, not]).
change(french, german).
change(do, no).
change(X, X).

alter([],[]).
alter([H|T], [X|Y]):-
        change(H, X),
        alter(T, Y).
```

- Note that this program is limited:
  - it would change "i do like you" into "i no like i",

Recursive mapping (continued)

- The program:

```
change(you, i).
change(are, [am, not]).
change(french, german).
change(do, no).
change(X, X).

alter([],[]).
alter([H|T], [X|Y]):-
        change(H, X),
        alter(T, Y).
```

- Note that this program is limited:
  - it would change “i do like you” into “i no like i”,
  - new rules would have to be added to the program to deal with such situations.

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: aless /2

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: $\mathsf{aless}/2$
  1. $\mathsf{aless}(\mathsf{book}, \mathsf{bookbinder})$ succeeds.

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: aless /2
  1. aless (book, bookbinder) succeeds.
  2. aless (ele p hant, ele v ator) succeeds.

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: aless /2
  1. aless (book, bookbinder) succeeds.
  2. aless (elephant, elevator) succeeds.
  3. aless (lazy, leather) is decided by aless (azy, eather).

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: aless/2
  1. aless(book, bookbinder) succeeds.
  2. aless(elephant, elevator) succeeds.
  3. aless(lazy, leather) is decided by aless(azy, eather).
  4. aless(same, same) fails.

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: aless /2
  1. aless (book, bookbinder) succeeds.
  2. aless (elephant, elevator) succeeds.
  3. aless (lazy, leather) is decided by aless (azy, eather).
  4. aless (same, same) fails.
  5. aless (alphabetic, alp) fails.

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: aless /2
  1. aless (book, bookbinder) succeeds.
  2. aless (elephant, elevator) succeeds.
  3. aless (lazy , leather) is decided by aless (azy, eather).
  4. aless (same, same) fails.
  5. aless (alphabetic , alp) fails.
- Use the predicate name/2 which returns the name of a symbol:

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

Comparing Structures

- Dictionary comparison (lexicographic comparison) of atoms: aless/2
  1. aless(book, bookbinder) succeeds.
  2. aless(elephant, elevator) succeeds.
  3. aless(lazy, leather) is decided by aless(azy, eather).
  4. aless(same, same) fails.
  5. aless(alphabetic, alp) fails.
- Use the predicate name/2 which returns the name of a symbol:

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

- The program:

```
aless(X, Y):-
      name(X, L), name(Y, M), alessx(L,M).

alessx([], [_|_]).
alessx([X|_], [Y|_]):- X < Y.
alessx([H|X], [H|Y]):- aless(X, Y).
```

Append

- We want to append two lists, i.e.

```
?-appendLists([a,b,c], [3,2,1], [a,b,c,3,2,1]
true
```

This illustrate the use of appendLists/3 for testing that a list is the result of appending two other lists.

Append

- We want to append two lists, i.e.

```
?-appendLists([a,b,c], [3,2,1], [a,b,c,3,2,1]
true
```

This illustrate the use of appendLists/3 for testing that a list is the result of appending two other lists.

- Other uses of appendLists/3:

Append

- We want to append two lists, i.e.

```
?- appendLists([a,b,c], [3,2,1], [a,b,c,3,2,1]
true
```

  This illustrate the use of appendLists/3 for testing that a list is the result of appending two other lists.

- Other uses of appendLists/3:

\- Total list computation:

```
?- appendLists([a, b, c], [3, 2, 1], X).
```

Append

- We want to append two lists, i.e.

```
?-appendLists([a,b,c], [3,2,1], [a,b,c,3,2,1]
true
```

  This illustrate the use of appendLists/3 for testing that a list is the result of appending two other lists.

- Other uses of appendLists/3:

- Total list computation:

```
?-appendLists([a, b, c], [3, 2, 1], X).
```

- Isolate:

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

Append

- We want to append two lists, i.e.

```
?-appendLists([a,b,c], [3,2,1], [a,b,c,3,2,1]
true
```

  This illustrate the use of appendLists/3 for testing that a list is the result of appending two other lists.

- Other uses of appendLists/3:

- Total list computation:

```
?-appendLists([a, b, c], [3, 2, 1], X).
```

- Isolate:

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

- Split:

```
?-appendLists(X, Y, [a, b, c, 3, 2, 1]).
```

```
        % the boundary condition
appendLists([ ], L, L).
        % recursion
appendLists([X|L1], L2, [X| L3]):-
     appendLists(L1, L2, L3).
```

Summary

- The recursive nature of structures (and in particular lists) gives a way to traverse them by recursive decomposition.

Summary

- The recursive nature of structures (and in particular lists) gives a way to traverse them by recursive decomposition.
- When the boundary is reached, the decomposition stops and the result is composed in a reverse of the decomposition process.

Summary

- The recursive nature of structures (and in particular lists) gives a way to traverse them by recursive decomposition.
- When the boundary is reached, the decomposition stops and the result is composed in a reverse of the decomposition process.
- This process can be made more efficient: introduce an extra variable in which the “result so far” is accumulated.

Summary

- The recursive nature of structures (and in particular lists) gives a way to traverse them by recursive decomposition.
- When the boundary is reached, the decomposition stops and the result is composed in a reverse of the decomposition process.
- This process can be made more efficient: introduce an extra variable in which the "result so far" is accumulated.
- When the boundary is reached this extra variable already contains the result, no need to go back and compose the final result.

Summary

- The recursive nature of structures (and in particular lists) gives a way to traverse them by recursive decomposition.
- When the boundary is reached, the decomposition stops and the result is composed in a reverse of the decomposition process.
- This process can be made more efficient: introduce an extra variable in which the "result so far" is accumulated.
- When the boundary is reached this extra variable already contains the result, no need to go back and compose the final result.
- This variable is called an accumulator.

Example: List Length

- Without accumulator:

```
% length of a list
 % boundary condition
 listlen([], 0).
 % recursion
 listlen([H|T], N):-
        listlen(T, N1),
        N is N1+1.
```

- With accumulator:

```
% length of a list with accumulators
 % call of the accumulator:
   listlen1(L, N):-
         lenacc(L, 0, N).
 % boundary condition for accumulator
   lenacc([], A, A).
 % recursion for the accumulator
   lenacc([H|T], A, N):-
          A1 is A + 1,
          lenacc(T, A1, N).
```

- With accumulator:

```
% length of a list with accumulators
 % call of the accumulator:
    listlen1(L, N):-
          lenacc(L, 0, N).
 % boundary condition for accumulator
    lenacc([], A, A).
 % recursion for the accumulator
    lenacc([H|T], A, N):-
           A1 is A + 1,
           lenacc(T, A1, N).
```

- Inside Prolog, for the query ?- listlen1 ([a, b, c], N):

```
lenacc([a, b, c], 0, N).
lenacc([b, c], 1, N).
lenacc([c], 2, N).
lenacc([], 3, N)
```

The return variable is shared by every goal in the trace.

Example: Reverse

- Without accumulators:

```
%% reverse
% boundary condition
  reverse1([],[]).
% recursion
  reverse1([X|TX], L):-
      reverse1(TX, NL),
      appendLists(NL, [X], L).
```

- With accumulators:

```
%% reverse with accumulators
% call the accumulator
  reverse2(L, R):-
      reverseAcc(L, [], R).
% boundary condition for the accumulator
  reverseAcc([], R, R).
% recursion for the accumulator
  reverseAcc([H|T], A, R):-
     reverseAcc(T, [H|A], R).
```

Summary

- Accumulators provide a technique to keep trace of the “result so far” (in the accumulator variable) at each step of computation, such that when the structure is traversed the accumulator contains “the final result”, which is then passed to the “output variable”.

Summary

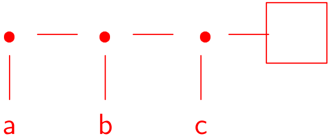
- Accumulators provide a technique to keep trace of the "result so far" (in the accumulator variable) at each step of computation, such that when the structure is traversed the accumulator contains "the final result", which is then passed to the "output variable".
- Now we consider a technique where we use a variable to hold "the final result" and the second to indicate a "hole in the final result", where more things can be inserted.

Summary

- Accumulators provide a technique to keep trace of the "result so far" (in the accumulator variable) at each step of computation, such that when the structure is traversed the accumulator contains "the final result", which is then passed to the "output variable".
- Now we consider a technique where we use a variable to hold "the final result" and the second to indicate a "hole in the final result", where more things can be inserted.
- Consider `[a, b, c | X]` - we know that this structure is a list up to a point (up to X). We call this an open list (a list with a "hole").

Summary

- Accumulators provide a technique to keep trace of the "result so far" (in the accumulator variable) at each step of computation, such that when the structure is traversed the accumulator contains "the final result", which is then passed to the "output variable".
- Now we consider a technique where we use a variable to hold "the final result" and the second to indicate a "hole in the final result", where more things can be inserted.
- Consider `[a, b, c | X]` - we know that this structure is a list up to a point (up to X). We call this an open list (a list with a "hole").



Using open lists

- Consider

```
?- X = [a, b, c | L], L = [d, e, f, g].
   X = [a, b, c, d, e, f, g],
   L = [d, e, f, g].
```

Using open lists

- Consider

```
?- X = [a, b, c | L], L = [d, e, f, g].
      X = [a, b, c, d, e, f, g],
      L = [d, e, f, g].
```

  - the result is the concatenation of the beginning of X (the list before the "hole") with L,

Using open lists

- Consider

```
?- X = [a, b, c | L], L = [d, e, f, g].
   X = [a, b, c, d, e, f, g],
   L = [d, e, f, g].
```

  - the result is the concatenation of the beginning of X (the list before the "hole") with L,
  - i.e. we filled the "hole",

Using open lists

- Consider

```
?- X = [a, b, c | L], L = [d, e, f, g].
      X = [a, b, c, d, e, f, g],
      L = [d, e, f, g].
```

  - the result is the concatenation of the beginning of X (the list before the “hole”) with L,
  - i.e. we filled the “hole”,
  - and this is done in one step!

Using open lists

- Consider

```
?- X = [a, b, c | L], L = [d, e, f, g].
      X = [a, b, c, d, e, f, g],
      L = [d, e, f, g].
```

  - the result is the concatenation of the beginning of X (the list before the “hole”) with L,
  - i.e. we filled the “hole”,
  - and this is done in one step!

- Now fill the hole with an open list:

```
?- X = [a, b, c | L], L = [d, e | L1].
      X = [a, b, c, d, e|L1],
      L = [d, e|L1].
```

Using open lists

- Consider

```
?- X = [a, b, c | L], L = [d, e, f, g].
      X = [a, b, c, d, e, f, g],
      L = [d, e, f, g].
```

  - the result is the concatenation of the beginning of X (the list before the "hole") with L,
  - i.e. we filled the "hole",
  - and this is done in one step!

- Now fill the hole with an open list:

```
?- X = [a, b, c | L], L = [d, e | L1].
      X = [a, b, c, d, e|L1],
      L = [d, e|L1].
```

  - the hole was filled partially.

- Now express this as a Prolog predicate:

```
diff_append1(OpenList, Hole, L):-
          Hole=L.
```

- Now express this as a Prolog predicate:

```
diff_append1(OpenList, Hole, L):-
        Hole=L.
```

i.e. we have an open list (OpenList), with a hole (Hole) is filled with a list (L):

```
?- X = [a, b, c, d | Hole],
           diff_append1(X, Hole, [d, e]).
   X = [a, b, c, d, d, e],
   Hole = [d, e].
```

- Note that when we work with open lists we need to have information (i.e. a variable) both for the open list and its hole.

- Note that when we work with open lists we need to have information (i.e. a variable) both for the open list and its hole.
- A list can be represented as the the difference between an open list and its hole.

- Note that when we work with open lists we need to have information (i.e. a variable) both for the open list and its hole.
- A list can be represented as the the difference between an open list and its hole.
- Notation: OpenList–Hole

- Note that when we work with open lists we need to have information (i.e. a variable) both for the open list and its hole.
- A list can be represented as the the difference between an open list and its hole.
- Notation: OpenList−Hole
  - here the difference operator − has no interpretation,

- Note that when we work with open lists we need to have information (i.e. a variable) both for the open list and its hole.
- A list can be represented as the the difference between an open list and its hole.
- Notation: OpenList−Hole
  - here the difference operator − has no interpretation,
  - in fact other operators could be used instead.

- Now modify the append predicate to use difference list notation:

```
diff_append2(OpenList-Hole, L):-
        Hole = L.
```

- Now modify the append predicate to use difference list notation:

```
diff_append2(OpenList-Hole, L):-
        Hole = L.
```

its usage:

```
?- X = [a, b, c, d | Hole]-Hole,
              diff_append2(X,[d, e]).
   X = [a, b, c, d, d, e]-[d, e],
   Hole = [d, e].
```

- Now modify the append predicate to use difference list notation:

```
diff_append2(OpenList-Hole, L):-
          Hole = L.
```

its usage:

```
?- X = [a, b, c, d | Hole]-Hole,
                diff_append2(X,[d, e]).
   X = [a, b, c, d, d, e]-[d, e],
   Hole = [d, e].
```

- Perhaps the fact that the answer is given as a difference list is not convenient.

- A new version that returns a(n open) list (with the hole filled) as the answer:

```
diff_append3(OpenList-Hole, L, OpenList):-
                                  Hole = L.
```

- A new version that returns a(n open) list (with the hole filled) as the answer:

```
diff_append3(OpenList-Hole, L, OpenList):-
                                        Hole = L.
```

its usage:

```
?- X = [a, b, c, d | Hole]-Hole,
              diff_append3(X,[d, e], Ans).
   X = [a, b, c, d, d, e]-[d, e],
   Hole = [d, e],
   Ans = [a, b, c, d, d, e].
```

- A new version that returns a(n open) list (with the hole filled) as the answer:

```
diff_append3(OpenList-Hole, L, OpenList):-
                                        Hole = L.
```

  its usage:

```
?- X = [a, b, c, d | Hole]-Hole,
               diff_append3(X,[d, e], Ans).
   X = [a, b, c, d, d, e]-[d, e],
   Hole = [d, e],
   Ans = [a, b, c, d, d, e].
```

- diff_append3 has

- A new version that returns a(n open) list (with the hole filled) as the answer:

```
diff_append3(OpenList-Hole, L, OpenList):-
                                        Hole = L.
```

  its usage:

```
?- X = [a, b, c, d | Hole]-Hole,
             diff_append3(X,[d, e], Ans).
   X = [a, b, c, d, d, e]-[d, e],
   Hole = [d, e],
   Ans = [a, b, c, d, d, e].
```

- diff_append3 has
  - a difference list as its first argument,

- A new version that returns a(n open) list (with the hole filled) as the answer:

```
diff_append3(OpenList-Hole, L, OpenList):-
                              Hole = L.
```

  its usage:

```
?- X = [a, b, c, d | Hole]-Hole,
           diff_append3(X,[d, e], Ans).
   X = [a, b, c, d, d, e]-[d, e],
   Hole = [d, e],
   Ans = [a, b, c, d, d, e].
```

- diff_append3 has
  - a difference list as its first argument,
  - a proper list as its second argument,

- A new version that returns a(n open) list (with the hole filled) as the answer:

```
diff_append3(OpenList-Hole, L, OpenList):-
                                Hole = L.
```

its usage:

```
?- X = [a, b, c, d | Hole]-Hole,
             diff_append3(X,[d, e], Ans).
   X = [a, b, c, d, d, e]-[d, e],
   Hole = [d, e],
   Ans = [a, b, c, d, d, e].
```

- diff_append3 has
  - a difference list as its first argument,
  - a proper list as its second argument,
  - returns a proper list.

- A further modification – to be systematic – for this version the arguments are all difference lists:

```
diff_append4(OL1-Hole1, OL2-Hole2, OL1-Hole2) :-
        Hole1 = OL2.
```

- A further modification – to be systematic – for this version the arguments are all difference lists:

```
diff_append4(OL1-Hole1, OL2-Hole2, OL1-Hole2) :-
        Hole1 = OL2.
```

and its usage:

```
?- X=[a,b,c|Ho]-Ho,
           diff_append4(X, [d,e,f|Hole2]-Hole2, Ans).
   X = [a, b, c, d, e, f|Hole2]-[d, e, f|Hole2],
   Ho = [d, e, f|Hole2],
   Ans = [a, b, c, d, e, f|Hole2]-Hole2.
```

- A further modification – to be systematic – for this version the arguments are all difference lists:

```
diff_append4(OL1-Hole1, OL2-Hole2, OL1-Hole2)
        Hole1 = OL2.
```

and its usage:

```
?- X=[a,b,c|Ho]-Ho,
              diff_append4(X, [d,e,f|Hole2]-Hole2,
   X = [a, b, c, d, e, f|Hole2]-[d, e, f|Hole
   Ho = [d, e, f|Hole2],
   Ans = [a, b, c, d, e, f|Hole2]-Hole2.
```

or, if we want the result to be just the list, fill the hole with the empty list:

```
?- X=[a,b,c|Ho]-Ho,
        diff_append6(X, [d,e,f|Hole2]-Hole2,
                                 Ans-[]).
   X = [a, b, c, d, e, f]-[d, e, f],
   Ho = [d, e, f],
   Hole2 = [],
```

- One last modification is possible:

```
append_diff(OL1-Hole1, Hole1-Hole2, OL1-Hol
```

- One last modification is possible:

```
append_diff(OL1-Hole1, Hole1-Hole2, OL1-Hol
```

its usage:

```
?- X=[a,b,c|H]-H,
      append_diff(X, [d,e,f|Hole2]-Hole2,
                          Ans-[]).
   X = [a, b, c, d, e, f]-[d, e, f],
   H = [d, e, f],
   Hole2 = [],
   Ans = [a, b, c, d, e, f].
```

Example: adding to back

- Let us consider the program for adding one element to the back of a list:

```
    % boundary condition
add_to_back(El,[],[El]).
    % recursion
add_to_back(El,[Head|Tail],[Head|NewTail):-
                add_to_back(El,Tail,NewTail).
```

Example: adding to back

- Let us consider the program for adding one element to the back of a list:

```
    % boundary condition
add_to_back(El,[],[El]).
    % recursion
add_to_back(El,[Head|Tail],[Head|NewTail]):-
                    add_to_back(El,Tail,NewTail).
```

- The program above is quite inefficient, at least compared with the similar operation of adding an element at the beginning of a list (linear in the length of the list – one goes through the whole list to find its end – versus constant – one step).

Example: adding to back

- Let us consider the program for adding one element to the back of a list:

```
      % boundary condition
add_to_back(El,[],[El]).
      % recursion
add_to_back(El,[Head|Tail],[Head|NewTail]):-
                  add_to_back(El,Tail,NewTail).
```

- The program above is quite inefficient, at least compared with the similar operation of adding an element at the beginning of a list (linear in the length of the list – one goes through the whole list to find its end – versus constant – one step).
- But difference lists can help - the hole is at the end of the list:

```
add_to_back_d(El,OpenList-Hole, Ans):-
    append_diff(OpenList-Hole, [El|ElHole]-ElHole
```

Problems with difference lists

- Consider:

```
?- append_diff([a, b] - [b], [c, d]-[d], L).
   false.
```

Problems with difference lists

- Consider:

```
?- append_diff([a, b] - [b], [c, d]-[d], L).
   false.
```

The above does not work! (no holes to fill).

Problems with difference lists

- Consider:

```
?- append_diff([a, b] - [b], [c, d]-[d], L).
   false.
```

  The above does not work! (no holes to fill).

- There are also problems with the occurs check (or lack there of):

```
empty(L-L).

?- empty([a|Y]-Y).
   Y = [a|**].
```

- − in difference lists is a partial function. It is not defined for [a, b, c]−[d] :

```
?- append_diff([a, b]-[c], [c]-[d], L).
   L = [a, b]-[d].
```

- − in difference lists is a partial function. It is not defined for [a, b, c]−[d] :

```
?- append_diff([a, b]-[c], [c]-[d], L).
   L = [a, b]-[d].
```

The query succeeds, but the result is not the one expected.

- $-$ in difference lists is a partial function. It is not defined for [a, b, c]−[d] :

```
?- append_diff([a, b]-[c], [c]-[d], L).
   L = [a, b]-[d].
```

  The query succeeds, but the result is not the one expected.

- This can be fixed:

```
append_diff_fix(X-Y, Y-Z, X-Z):-
        suffix(Y, X),
        suffix(Z, Y).
```

- − in difference lists is a partial function. It is not defined for [a, b, c]−[d] :

```
?- append_diff([a, b]-[c], [c]-[d], L).
   L = [a, b]-[d].
```

  The query succeeds, but the result is not the one expected.
- This can be fixed:

```
append_diff_fix(X-Y, Y-Z, X-Z):-
        suffix(Y, X),
        suffix(Z, Y).
```

  however, now the execution time becomes linear again.