Logic Programming

Lists. Recursion

- Lists
- Recursion

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)

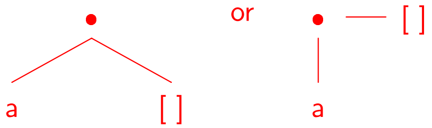
- "Base case": [ ] – the empty list.
- "General case" : .(h, t) – the nonempty list, where:
  - h - the head, can be any term,

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

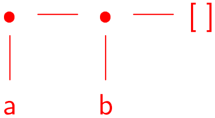
- .(a, [ ]) is represented as



"tree representation" *"vine representation"*

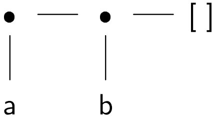
List representations (cont'd)

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

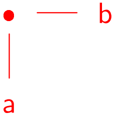


List representations (cont'd)

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

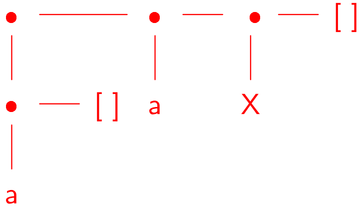


- .(a, b) is *not* a list, but it is a legal Prolog structure, represented as



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**.

Exercises

- Define predicates in Prolog for:

Exercises

- Define predicates in Prolog for:
  1. The length of a list

Exercises

- Define predicates in Prolog for:
  1. The length of a list
  2. The sum of elements of a list

Exercises

- Define predicates in Prolog for:
    1. The length of a list
    2. The sum of elements of a list
    3. The reverse of a list

Exercises

- Define predicates in Prolog for:
  1. The length of a list
  2. The sum of elements of a list
  3. The reverse of a list
  4. The list of elements on even positions

Exercises

- Define predicates in Prolog for:
  1. The length of a list
  2. The sum of elements of a list
  3. The reverse of a list
  4. The list of elements on even positions
  5. The concatenation of two lists.