Chapter 16

Logic Programming

Topics

- Examples
  - Execution trace
  - Controlling backtracking
- Lists

Lists

- One of the most important Prolog data structures.
- A list is an ordered sequence of zero or more terms written between square brackets and separated by commas.

```
[alpha,beta,gamma,delta]
[1,2,3,go]
[(2+2),in(austin,texas),-4.356,X]
[[a,list,within],a,list]
```


Lists

- The elements of a list can be Prolog terms of any kind, including other lists.
  - The element `[a]` is not equivalent to the atom a.
- The empty list is written `[]`.
- List can be constructed or decomposed through unification.
  - An entire list can match a single variable

Unify	With	Result
`[a,b,c]`	`X`	`X = [a,b,c]`

Lists

- Corresponding elements of two lists can be unified one by one.

Unify	With	Result
`[X,Y,Z]`	`[a,b,c]`	`X=a, Y=b, Z=c`
`[X,b,Z]`	`[a,Y,c]`	`X=a, Y=b, Z=c`

- This also applies to lists or structures embedded within lists.

Unify	With	Result
`[[a,b],c]`	`[X,Y]`	`X=[a,b], Y=c`
`[a(b),c(X)]`	`[Z,c(a)]`	`X=a, Z=a(b)`

Lists: head & tail

- Any list can be divided into head and tail by the symbol |
  - The head of a list is the first element
  - The tail is a list of the remaining elements (and can be empty).
  - *The tail of a list is always a list; the head of a list is an element.*

Lists: examples

- Every nonempty list has a head and a tail
  `[a|[b,c,d]] = [a,b,c,d]`
  `[a|[]] = [a]`
- The term `[X|Y]` unifies with any nonempty list, instantiating `X` to the head and `Y` to the tail

Unify	With	Result
`[X|Y]`	`[a,b,c,d]`	`X=a, Y=[b,c,d]`
`[X|Y]`	`[a]`	`X=a, Y=[]`

Lists: examples

Unify	With	Result
`[X,Y|Z]`	`[a,b,c]`	`X=a, Y=b, Z=[c]`
`[X,Y|Z]`	`[a,b,c,d]`	`X=a, Y=b, Z=[c,d]`
`[X,Y,Z|A]`	`[a,b,c]`	`X=a, Y=b, Z=c, A=[]`
`[X,Y,Z|A]`	`[a,b]`	fails
`[X,Y,a]`	`[Z,b,Z]`	`X=Z=a, Y=b`
`[X,Y|Z]`	`[a|W]`	`X=a, W=[Y|Z]`

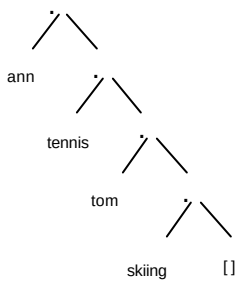
Lists: internal representation

- The previous representation is only the external appearance of lists.
- All structured objects in Prolog are trees.
  - The head and the tail are combined into a structure by a special functor `'.'`

    `.(Head, Tail)`
- Example:

`[ann,tennis,tom,skiing]`
`.(ann,.(tennis,.(tom,.(skiing,[])))).`

Lists: internal representation



- Both notations can be used.
- The square bracket notation is normally preferred.
- Internally, they are represented as binary trees.

Lists: recursion

- To fully exploit the power of lists:
  - A way to work with lists elements without specifying their positions in advanced.
- A repetitive procedure that will work their way along a list, searching for a particular element or performing some operation on every element encountered.
  - Repetition is expressed in Prolog by using *recursion*.

Lists: recursion

- In order to solve a problem, perform some action and then solve a similar problem of the same type using the same procedure.
- The process terminates when the problem becomes so small that the procedure can solve it in one step without calling itself again.
- Some common operations on lists: membership, concatenation, adding an item to a list, etc

Lists: membership

- A predicate `member(X,L)` succeeds if `X` is an element of the list `L`.
- Examples:

```
?- member(b,[a,b,c]).
   yes
?- member(b,[a,[b,c]]).
   no
?- member([b,c],[a,[b,c]]).
   yes
```


Lists: membership

- What can we say about the membership relation?
- In general, this relationship can be based on the following observation:

  X is a member of L if either

  (1) X is the head of L, or

  (2) X is a member of the tail of L.

Lists: membership

- Identify two special case that are not repetitive
  - If `L` is empty, fail with no further action (nothing is a member of the empty list).
  - If `X` is the first element of `L`, succeed with no further action (the element was found).
- To solve the first special case: make sure in all clauses that the second argument is something that will not unify with an empty list.

Lists: membership

- The second argument should be a list that has both a head and a tail.
  - The second special case (a simple clause):

```
member(X,[X|Tail]).
```

- The recursive part ("X is a member of L if X is a member of the tail of L") can be expressed as:

```
member(X,[Head|Tail]):-
            member(X,Tail).
```


Lists: concatenation

- The predicate concatenate or append combines to lists into a single list.
- Examples:

```
?- concatenate([a,b,c],[d,e,f],X).
   X = [a,b,c,d,e,f].
```

- Can we use '|'? `[[a,b,c]|[d,e,f]]`

  is equivalent to `[[a,b,c],d,e,f]`.
- Strategy: work through the first list element by element, adding elements one by one to the second list.

Lists: concatenation

- The definition `concatenation(L1,L2,L3)` will have again two cases, depending on the first argument, `L1`:

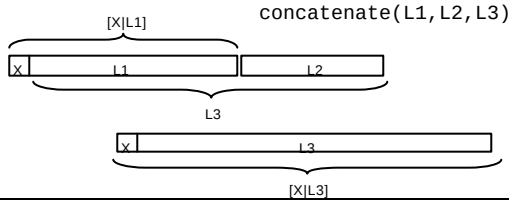
  (1) Since the first list will be shortened, it will eventually become empty. So, if the first argument is an empty list then the second and the third arguments must be the same list.

```
concatenate([],L,L).
```

Lists: concatenation

(2) If the first argument is a non-empty list then it has a head and a tail. The new list have the same head and the concatenation of the tail with the second list.

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



Lists: concatenation

- Examples:

```
?- concatenate([a,[b,c],d],[a,[],b],X).
   X = [a,[b,c],d,a,[],b]
?- concatenate([a,b,c],X,[a,b,c,d,e,f]).
   X = [d,e,f]
?- concatenate(X,[d,e,f],[a,b,c,d,e,f]).
   X = [a,b,c]
?- concatenate(X,Y,[a,b,c,d]).
   X = []  Y = [a,b,c,d]
   X = [a]  Y = [b,c,d]        X = [a,b]  Y = [c,d]
   X = [a,b,c]  Y = [d]        X = [a,b,c,d]  Y = []
```


Lists: adding an item

- To add an item to a list, it is easier to put the new item in front of the list so that it becomes the new head.

```
add(X,L,[X|L]).
```

- Examples:

```
?- add(0,[1,2,3],L).
   L = [0,1,2,3]
?- add(X,[b,c,d],[a,b,c,d]).
   L = a
```


Lists: deleting an item

- The delete operation `delete(X,L,L1)` deletes an item X from a list L, where L1 is equal to the list L with the item X removed.

(1) If X is the head of the list then the result after the deletion is the tail of the list.

```
delete(X,[X|Tail],Tail).
```

(2) If X is in the tail then it is deleted from there.

```
delete(X,[Y|Tail],[Y|Tail1]):-
                    delete(X,Tail,Tail1).
```


Lists: deleting an item

- `Delete`, like `member`, is also non-deterministic in nature.
  - If there are several occurrences of X in the list then delete will be able to delete anyone of them by backtracking.
  - Each alternative execution will only delete one occurrence of X, leaving the others untouched.

```
?- delete(a,[a,b,a,a],L).
L = [b,a,a];
L = [a,b,a];
L = [a,b,a];
no
```


Lists: counting list elements

- This is the recursive algorithm to count elements of a list:
  - If the list is empty, it has 0 elements.

```
list_length([],0).
```

  - Otherwise, skip over the first element, count the number of elements remaining and add 1.

```
list_length([_|Tail],K) :-
                list_length(Tail,J).
                K is J+1.
```

Lists: reversing a list

- Recursive algorithm for reversing the order of elements in a list:
  (1) Split the original list into a head and tail.
  (2) Recursively reverse the tail of the original list.
  (3) Make a list whose only element is the head of the original list.
  (4) Concatenate the reversed tail of the original list with the list created in step 3.

Lists: reversing a list

- The list gets shorter every time, the limiting case is an empty list, which Prolog must return unchanged.

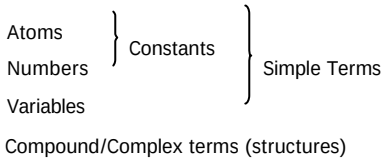
```
reverse([],[]).
reverse([Head|Tail],Result) :-
   reverse(Tail,ReverseTail),
   concatenate(ReverseTail,[Head],Result).
```

- Example:

```
?- reverse([a,b,c,d],X).
   X = [d,c,b,a]
```


Terms

- All Prolog statements are constructed from terms.



Family Facts

```
parent(pam,tom).        female(pam).
parent(tom,bob).        male(tom).
parent(tom,liz).        male(bob).
parent(bob,ann).        female(liz).
parent(bob,pat).        female(ann).
parent(pat,jim).        female(pat).
                        male(jim).
```


Controlling backtracking: example

- Consider the double step function. The relation between X and Y can be specified by three rules:
  - Rule 1: if X < 3 then Y = 0
  - Rule 2: If 3 ≤ X and X < 6 then Y = 2
  - Rule 3: if 6 ≤ X then Y = 4

Controlling backtracking: experiment 1

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

- Question: `?- f(1,Y), 2 < Y.`
- The first goal `f(1,Y), Y` becomes instantiated to `0`.
- The second goal becomes `2 < 0` which fails, and so does the whole goal list.
- Before admitting that the goal list is not satisfiable, Prolog tries, through backtracking, two useless alternatives.
- The three rules about the `f` relation are mutually exclusive so that one of them at most will succeed.

Controlling backtracking: experiment 2

```
f(X,0) :- X < 3, !.
f(X,2) :- X < 6, !.
f(X,4).
```

- Equivalent to these three rules:

  if X <3 then Y =0,

  otherwise if X < 6 then Y = 2,

  otherwise Y = 4

Examples using cut

- Computing maximum

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

- Single-solution membership

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

- Classification into categories

```
class(X, fighter) :- beat(X,_),beat(_,X),!.
class(X,winner) :- beat(X,_), !.
class(X, sportsman) :- beat(_,X).
```