Lecture 12: Working with lists

Deep lists. Difference lists. Applications. The maze problem

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

May 17, 2021

Outline of this lecture

1. Review of Prolog datatypes and type recognizers
   - Special datatypes
     - lists
     - tuples
2. Lists
   - Working with deep lists
   - An alternative representation of lists: Difference lists
     - Efficiency issues
     - Applications of difference lists
3. The maze problem

Data structures in Prolog

Recap: In Prolog, terms are the only datatype

Prolog has only one datatype: **terms** (see Lecture 8).

- A term is either:
  - an atomic term. There are 3 kinds of atomic terms:
    - atom (or function name): a name which starts with lowercase letter, or is delimited by quotes. Exemples: `car`, `'I am Sam'`
    - number: `1.23` (floating point), `-283416043388` (arbitrary-size integer)
    - string: `"Mary had a little lamb"`
    - the symbol `[]` for the empty list.
  - variable: name which begins with uppercase letter or with _. Variables are placeholders for terms. Examples: `X`, `_X`
  - compound term: $f(term_1, \ldots, term_n)$ where f is an atom and $term_1, \ldots, term_n$ are terms.

$$term \;::=\; atomic \;\mid\; variable \;\mid\; f\,(term_1, \ldots, term_n)$$

The following term recognizers are predefined:

- `atomic`(t): holds if t is an atomic term.
- `atom`(t): holds if t is atom.
- `number`(t): holds if t is number: floating-point or integer.
- `float`(t): holds if t is a floating-point number.
- `integer`(t): holds if t is an integer.
- `string`(t): holds if t is a string.
- `compound`(t): holds if t is a compound term.
- `var`(t): true if t is currently a free variable.
- `nonvar`(t): true if t is currently not a free variable.

In Prolog, atom has three meanings: it can be

1. a function symbol
2. a predicate symbol, or
3. an atomic formula $p(t_1, \ldots, t_n)$ where p is a predicate symbol and $t_1, \ldots, t_n$ are terms.

Example

```
snowman(olaf).
melts(X) :- snowman(X).
```

This program contains

function symbol `olaf`

predicate symbols `snowman` and `melts`

atomic formulas `snowman(X)` and `melts(X)`

All these things are atoms.

Terms with special syntax (recap)

Arithmetic expressions, lists and tuples

- Arithmetic expressions: t_1 *op* t_2 instead of $'op'(t_1, t_2)$. We can write `X+3*4` instead of `'+'(X,'*'(3,4))`

  The arithmetic operations are predefined: $+, *, /, -$, etc.
  We can evaluate an arithmetic expression *expr* with `X is` *expr*.

- Lists are terms defined by grammar

  $$list \; ::= \; \texttt{[]} \; \mid \; \texttt{'[|]'}(term, list)$$

  The list constructor `'[|]'` is a predefined function symbol.
  We can write `[a,b,c]` instead of `'[|]'(a,'[|]'(b,'[|]'(c,[])))`.

- Tuples are terms defined by grammar:

  $$tuple \; ::= \; \texttt{','}(term_1, term_2) \; \mid \; \texttt{','}(term, tuple)$$

  The pair constructor `','` is a predefined function symbol.
  We can write `(a,b,c)` instead of `','(a,','(b,c))`

Working with lists and tuples

Both lists and tuples can be taken apart by unification.

- Splitting a non-empty list into head(s) and tail:

```
?- [H|T]=[a,b,c].
H=a,
T=[b,c].
```

```
?- [H1,H2|T]=[a,b,c].
H1=a,
H2=b,
T=[c].
```

- Splitting a tuple into first component(s) and rest.

```
?- (F,R)=(a,b,c).
F=a,
R=(b,c).
```

```
?- (F1,F2,R)=(a,b,c).
F1=a.
F2=b,
R=c.
```

```
?- a = (a).
true.
```

```
?- (a,(b))=(a,b).
true.
```

Working with lists and tuples

Both lists and tuples can be taken apart by unification.

- Splitting a non-empty list into head(s) and tail:

```
?- [H|T]=[a,b,c].
H=a,
T=[b,c].
```

```
?- [H1,H2|T]=[a,b,c].
H1=a,
H2=b,
T=[c].
```

- Splitting a tuple into first component(s) and rest.

```
?- (F,R)=(a,b,c).
F=a,
R=(b,c).
```

```
?- (F1,F2,R)=(a,b,c).
F1=a.
F2=b,
R=c.
```

```
?- a = (a).
true.
```

```
?- (a,(b))=(a,b).
true.
```

Remark: There are no tuples with one component: `(`*term*`)` coincides with *term*.

In Prolog, lists can be nested one into another.

- A deep list is a list whose elements are either deep lists, or atomic terms. Formally:

  $$dlist \ ::= \ [\,] \ \mid \ [\,h \mid dlist\,] \quad \text{where}$$
  $$h \ ::= \ atom \ \mid \ number \ \mid \ string \ \mid \ dlist$$

- A deep list which does not contain another list as an element is called simple, or shallow. Formally:

  $$shlist \ ::= \ [\,] \ \mid \ [\,at \mid shlist\,] \quad \text{where}$$
  $$at \ ::= \ atom \ \mid \ number \ \mid \ string$$

Examples of deep lists

```
L1 = [1,2,3,[4]]
L2 = [[1],[2],[3],[4,5]]
L3 = [[],2,3,4,[5,[6]],[7]]
L4= [alpha,2,[beta],[gamma,[8]]]
```

Deep lists are special lists ⇒ all operations on lists work on deep lists too: `member`, `length`, `reverse`

We wish to add the following operations which are specific to deep lists:

1. `depth(L,R)` binds `R` to the depth of deep list `L`:
2. `flatten(DL,FL)` flattens deep list `DL` into a shallow list `FL`.
3. `heads(DL,Hs)` returns all elements which are at the head of a shallow list in `DL`.
4. `member1(X,DL)` holds if `X` occurs, at any depth, as an element of `DL`.
5. `member2(X,DL)` holds if `X` is non-list which occurs, at any depth, as an element of `DL`.

```
depth([],1).
depth([[]|T],R):-!,depth(T,R1),R is max(2,R1).
depth([H|T],R):-atomic(H),!,depth(T,R).
depth([H|T],R):-depth(H,R1),
                depth(T,R2),
                R is max(R1+1,R2).
flatten([],[]).
flatten([[]|T],FL):-!,flatten(T,FL).
flatten([H|T],[H|FL]):- atomic(H),!,
                        flatten(T,FL).
flatten([H|T],FL):- flatten(H,FL1),
                    flatten(T,FL2),
                    append(FL1,FL2,FL).
```

Note the usage of the cut operator (`!`) to simplify the implementation and make it more efficient.

Implement the predicates `heads`, `member1` and `member2`:

```
?- heads([[1,[2,3]],[[],[[4,5],6,[7]]]],L).
L = [1,2,4,7].

?- member1([2,3],[1,[2,3],4]).
true.

?- member2(3,[1,[2,[[],3,[4,5]]],6]).
true.

?- member2([4,5],[1,[2,[[],3,[4,5]]],6]).
false.
```

A list is accessed through its head and tail $\Rightarrow$ accessing the n-th element is slow (n steps): we must access all its elements before the n-th

- Open list = alternative way to represent a list in Prolog, that lets us access the end of a list easier.

  $$\mathit{openList} \ ::= \ [\,\mathit{term}_1, \ldots, \mathit{term}_n \,|\, H\,]$$

  where H is a free variable, and $\mathit{term}_1, \ldots, \mathit{term}_n$ are terms.
  - H acts like a pointer to the end of the list.
  - by instantiating `X` with a list, we extend the open list to a true list.

```
?- L=[1,2,3|H],H=[4,5].
L=[1,2,3,4,5].
```

diffList `::=` `dList(`*openList*, *H*`)`

where *openList* is an open list $[term_1, \ldots, term_n|H]$ with free variable H. *diffList* represents the list `[`$term_1, \ldots, term_n$`]` as a difference.

Appending difference lists

```
dAdd(dList(OL1,H1),dList(OL2,H2),dList(OL1,H2)):-
    H1=OL2.
```

Runtime complexity: $O(1)$. Note that `append(L1,L2,L)` has runtime complexity $O(n)$ where n is the length of list `L1`.

Example

```
?- dAdd(dList([1,2|H1],H1),dList([3,4|H2],H2),DL).
H1 = [3,4|H2],
DL = dList([1,2,3,4|H2],H2).
```

```
addToEnd(dList(OL,H),E,OL):-H=[E].
```

Runtime complexity: $O(1)$.

Example

```
?- addToEnd(dList([1,2,3,4,5,6|H],H),7,R).
H = [7],
R = [1,2,3,4,5,6,7].
```

```
member_open(_,dList(OL,H)):-
   unify_with_occurs_check(OL,H),!,fail.
member_open(X,dList([X|_],_)).
member_open(X,dList([_|OL],H)):-
       member_open(X,dList(OL,H)).
```

Example

```
?- member_open(X,dList([1,2|H],H)).
X=1 ;
X=2 ;
false.
```

Remark

By instantiating the free variable of an difference list, we destroy it: the open list becomes an ordinary list.

Remarks about unification in SWI-Prolog

In logic programming, the attempt to unify `X` with a non-variable term `t` which contains `X` fails – because of the variable-occur check test. We can check this fact with the built-in predicate `unify_with_occurs_check`:

```
?- unify_with_occurs_check(X,f(X)).
false.
```

SWI-Prolog allows to unify `X` with a non-variable term `t` which contains `X`. For example:

```
?- X = f(1,X),writeln(X).
@(S_1,[S_1=f(1,S_1)])
X = f(1, X).
```

is a weird notation to indicate that the result of unifying `X` with `f(1,X)` is the infinite term `f(1,f(1,f(1,...)))`

btree `::= nil | bt(`*integer*, *btree*, *btree*`)`

Main idea: use difference lists for fast concatenation of traversals of left- and right subtree.

```
inorder(BT,L) :- dInorder(BT,dList(L,H)),H=[].
dInorder(nil,dList(H,H)).
dInorder(bt(N,BT1,BT2),DL) :-
   dInorder(BT1,DL1),
   dInorder(BT2,DL2),
   dAdd(DL1,dList([N|H],H),DL3),
   dAdd(DL3,DL2,DL).
```

```
?- inorder(bt(2,bt(1,nil,nil),bt(3,nil,nil)),L).
L = [1,2,3].
```

1. Define the predicate `preorder(BT,L)` that binds `L` to the list of values in the nodes of the binary tree `BT`, by preorder traversal.
2. Define the predicate `postorder(BT,L)` that binds `L` to the list of values in the nodes of the binary tree `BT`, by postorder traversal.
3. Define the flattening predicate `flatten(DL,FL)` with difference lists.

Difference lists are an alternative representation of lists in Prolog:

$diffList$ ::= `dList([`$term_1, \ldots, term_n$`|`H`],`H`)`

- Represents the list `[`$term_1, \ldots, term_n$`]`. For example:
  `dList([a,b,c|H],H)` represents the list `[a,b,c]`
  `dList(H,H)` represents the empty list `[]`
- The free variable H is like a pointer to the end of the list.

The following operations with difference lists take constant time:

- append difference lists (predicate `dAdd`): $O(1)$
  built-in `append(L,L2,R)`: $O(n)$
- adding an element to end of difference list (`addToEnd`): $O(1)$
  adding an element to end of list `L`: $O(n)$

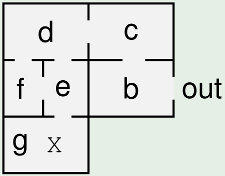
where n is length of `L`.

A person x is placed in a building with many rooms, connected by doors. One room has an exit door from the building.

Q1: How can x find a way out from the building?

Q2: How can x find the shortest way out from the building?

Example




```
door1(b,out).
door1(b,c).
door1(c,d).
door1(d,e).
door1(e,f).
door1(e,g).
door1(d,f).
```

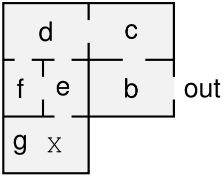
Possible answer 1: [g, e, f, d, c, b, out]

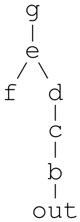
Possible answer 2: [g, e, d, c, b, out]

The maze problem

This is a typical search problem.

- We must find a trail, which is a list $[a_1, a_2, \ldots, a_n]$ from the point of departure a_1 to the destination a_n, such that, every two consecutive rooms a_i, a_{i+1} are connected by a door.
  - In our scenario: $a_1 = \texttt{g}$, $a_n = \texttt{out}$
  - Nontermination (cycles) can be avoided by keeping track ot the rooms already visited, to avoid visiting them again.
- Shortest way out = shortest trail from location of `X` to `out`.
  - can be found by breadth-first search:





To answer the first question (finding a way out from the building), we can define a predicate `go(X,Y,Trail)` which binds `Trail` to a trail from the location `X` to location `Y`:

- We use an accumulator `A` to accumulate in a list the rooms visited so far.

```
% initially, the only visited room is the initial room X
go(X,Y,Trail) :- goAcc(X,Y,Trail,[X]).
% we stop when we reach destination, that is, X=Y
goAcc(X,X,Trail,Acc) :- reverse(Acc,Trail).
% we move from X to Z if Z was not visited
% and if there is a door from X to Z
goAcc(X,Y,Trail,Acc) :-
    (door1(X,Z);door1(Z,X)),
    not(member(Z,Acc)),
    goAcc(Z,Y,Trail,[Z|Acc]).
```

The existence of a door from `X` to `Y` is represented be the fact `door1(X,Y).`

- We want this relation to be symmetric: if there is a door from `X` to `Y`, there is also a door from `Y` to `X`
  - To avoid nontermination of computing with symmetric relations, (see Lecture 10), we can define

```
door(X,Y):- door1(X,Y).
door(X,Y):- door1(Y,X).
...
goAcc(X,Y,Trail,Acc) :-
    door(X,Z),
    not(member(Z,Acc)),
    goAcc(Z,Y,Trail,[Z|Acc]).
```

The existence of a door from `X` to `Y` is represented be the fact `door1(X,Y)`.

- We want this relation to be symmetric: if there is a door from `X` to `Y`, there is also a door from `Y` to `X`
  - To avoid nontermination of computing with symmetric relations, (see Lecture 10), we can define

```
door(X,Y):- door1(X,Y).
door(X,Y):- door1(Y,X).
...
goAcc(X,Y,Trail,Acc) :-
    door(X,Z),
    not(member(Z,Acc)),
    goAcc(Z,Y,Trail,[Z|Acc]).
```

    SWI-Prolog allows to write `(door1(X,Z);door1(Z,X))` instead of `door(X,Z)`.

    - The intended reading of “`;`” is “or”.

The trail is accumulated in reverse order in `Acc` ⇒ when we reach destination, we must instantiate `Trail` with the reverse of `Acc`

- reversing `Acc` with n elements takes $O(n)$ time
- We can avoid this problem if, instead of an accumulator, we use a difference list ⇒ a more efficient version of predicate `go`:

```
goV2(X,Y,Trail):-
   goDiffList(X,Y,Trail,dList(H,H)).
goDiffList(Y,Y,Trail,DL):-
   addToEnd(DL,Y,Trail).
goDiffList(X,Y,Trail,DL):-
   (door1(X,Z);door1(Z,X)),
   not(member_open(Z,DL)),
   dAdd(DL,dList([X|H],H),DL1),
   goDiffList(Z,Y,Trail,DL1).
```

```
findAll(Term,+Query,-L)
```

is a predefined second-order predicate of SWI-Prolog: It binds the free variable `L` to the list of all terms `Term`θ where θ is a computed answer of `Query`.

Examples

% find all rooms connected by a door with room `g`

```
?- findall(X,(door1(f,X);door1(X,f)),L).
L = [d,e].
```

% find all rooms connected by a trail of length 3 with room `f`

```
?- findall(X,(goV2(f,X,Trail),length(Trail,3)))
L = [e,c,g,d].
```

We will use `findall` to implement predicate `goBF(X,Y,Trail)` that binds `Trail` to a shortest trail from `X` to `Y`, using breadth-first search.

To find a shortest trail (or path) from `X` to `Y` we proceed as follows:

- Starting from `X`, we generate all paths of length 0, then all paths of length 1, and so on.
  - The paths of length $n > 0$ are produced by extending those of length $n - 1$ with one more element. We will see how to do so with the `findall` operator.
- We stop as soon as we find a path from `X` to `Y`.

```
goBF(X,Y,Trail) :- goBFAux([[X]],Y,R),reverse(R,Trail).
goBFAux([[Y|Xs]|_],Y,[Y|Xs]):-!.
goBFAux([[Lf|Xs]|Rs],Y,Trail):-
     findall([Z,Lf|Xs],
             ((door1(Lf,Z);door1(Z,Lf)),
              not(member(Z,[Lf|Xs]))),
             ZRs),
     append(Rs,ZRs,NewRs),
     goBFAux(NewRs,Y,Trail).
```

Notes on the implementation of `goBF(X,Y,Trail)`

`goBF(X,Y,Trail)` has input arguments `X` (the start of search) and `Y` (the destination), and binds `Trail` to a shortest trail form `X` to `Y`:

- if such a trail does not exist, the predicate returns `false`.

The trail is computed by breadth-first search, implemented by the auxiliary predicate `goBFAux(Rs,Y,Trail)` which takes as inputs

- `Rs`: the list of branches of the breadth-first traversal tree with root `X`. In `Rs`, every branch is represented by the list of nodes from a leaf node to the root.
- `Y`: the destination node.

and binds `Trail` to a shortest trail from `X` to `Y`. This is obtained by reversing the first branch added to `Rs`, which ends with `Y`.

Notes on the implementation of `goBF(X,Y,Trail)`

`goBF(X,Y,Trail)` has input arguments `X` (the start of search) and `Y` (the destination), and binds `Trail` to a shortest trail form `X` to `Y`:

- if such a trail does not exist, the predicate returns `false`.

The trail is computed by breadth-first search, implemented by the auxiliary predicate `goBFAux(Rs,Y,Trail)` which takes as inputs

- `Rs`: the list of branches of the breadth-first traversal tree with root `X`. In `Rs`, every branch is represented by the list of nodes from a leaf node to the root.
- `Y`: the destination node.

and binds `Trail` to a shortest trail from `X` to `Y`. This is obtained by reversing the first branch added to `Rs`, which ends with `Y`.

Every recursive call of `goBFAux(Rs,X,Trail)` removes the first branch `Br`, from `Rs`, computes the list of `ZRs` of all branches produces by extending `Br` with a room `Z` not visited before, and append `ZRs` to the end of list `Rs`.