Practical Work of Languages, Technologies, and Paradigms of Programming

Part III
Logic Programming
2018-2019

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Practice 8: Introduction to lists in Prolog

Contents

1 Main aim of this practice **2**

2 Lists in Prolog **2**
2.1 A simple representation for lists 2
2.2 Flattening a list. 5
2.3 The last element of a list. 6
2.4 Sublists operations. 7
2.5 Reversing a list: accumulation parameters and tail recursion. 7
2.6 Final exercises with lists 8

1 Main aim of this practice

The main aim of this practice is to introduce the use of lists in Prolog.

2 Lists in Prolog

Intuitively, a list is a sequence of zero or more elements. In Prolog, the internal representation of lists can be seen as a special case of the representation of terms. In this context, a list can be define inductively as follows:

1. `[]` is an (empty) list;
2. `.(E,L)` is a (non-empty) list if `E` is an element and `L` is a list. The element `E` is known as the head whereas `L` is the tail of the list.

In a list, the function symbol "." can be understood as a binary operator that *constructs* the list `.(E,L)` *appending* `E` to the beginning of the list `L`. By analogy with other languages like Haskell, we could say that "`[]`" and "`.(_,_)`" are the constructors in the domain of lists. However, the choice of these constructor symbols is completely arbitrary and other ones could have been chosen instead; any constant symbol and function symbol should be also valid, for instance: "*nil*" or "*cons*". Our preference for the first two is that some systems Prolog, in particular SWI-Prolog, use this notation as an internal representation for lists.

Figure 1 shows the list composed by `1`, `2`, and `3` (i.e. the list `.(1,.(2,.(3,[])))` represented as the corresponding term (a binary tree).

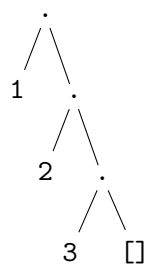

Figure 1: A list of integers.

Construct the visual representation in the form of binary terms of the following lists:

```
.(a,.(b,.(c,.(d,[ ]))))
.(1,.(.(juan,.(pedro,[ ])),.(2,[ ])))
.([ ],.(.(a,.(b,[ ])),.(.(1,.(2,.(3,[ ]))),[ ])))
.([ ],[ ])
```

Which is the most general unifier of `[E|L]` and `[2,2]`?

2.1 A simple representation for lists

The representation of the lists using the constructors "`[]`" and "`.` " can become truly confusing For this reason, the Prolog language provides an alternative, more friendly, notation to represent lists.

In the alternative notation, the binary prefix constructor "`.(_, _)`" is replaced by the infix operator "`[_|_]`". This allows to represent the list `.(E,L)` in infix notation, specifically as a structure `[E|L]`; that is, with the first element `E` separated by the symbol `|` from the remaining list `L`, which contains all the elements of the list except the first one.

In order to appreciate the difference of this representation of lists, it suffices to see how are now denoted the lists of previous example:

```
[a|[b|[c|[d|[ ]]]]],
[1|[[juan|[pedro|[ ]]]|[2|[ ]]]],
[[ ]|[[a|[b|[ ]]]|[[1|[2|[3|[ ]]]]|[ ]]]],
[[ ]|[ ]]
```

This notation is not yet readable enough, so Prolog admits and understands the following abbreviations:

1. `[`e_1`|[`e_2`|[`e_3`|...[`e_n`|L]]]]` is abbreviated as `[`e_1`, `e_2`, `e_3`, ..., `e_n`|L]`; and
2. `[`e_1`, `e_2`, `e_3`, ..., `e_n`|[ ]]` is abbreviated as `[`e_1`, `e_2`, `e_3`, ..., `e_n`]`

It is important to note that, with this new simplification, the list `[a|[b|[c|[d|[]]]]]` can be represented in many equivalent forms, being the simplest one `[a,b,c,d]`. In any case, any of the following representations would be equally equivalent: `[a,b,c,d|[]]`, or `[a,b,c|[d]]`, `[a,b|[c,d]]`, or even `[a|[b,c,d]]`.

Note that a list does not necessarily have to end with the empty list. In fact, in many cases a variable is used as a pattern that later will unify with the rest of another list. For example, in the list `[a, b|L]`, the variable `L` is likely to unify with the rest of any list whose first two elements are `a` and `b`. With these new possibilities, the lists of the examples can be finally denoted as:

```
[a,b,c,d],
[1,[juan,pedro],2],
[[],[a,b],[1,2,3]],
[[],[]]
```

We can appreciate a remarkable improvement in readability.

Like Haskell, Prolog allows retrieving information within the lists by adjusting patterns (in this case, bidirectional), as we will practice in the following exercises.

Exercise 1 *Edit a Prolog program and introduce the following facts:*

```
countTo(1,[1]).
countTo(2,[1,2]).
countTo(3,[1,2,3]).
countTo(4,[1,2,3,4]).
```

Load the code and write the queries:

```
?- countTo(X,[_,_,_,Y]).
?- countTo(X,[_,_,_|Y]).
```

In each case, is `Y` *an element or a list?*

Exercise 2 *Try the following queries:*

```
?- countTo(4,[H|T]).
?- countTo(4,[H1,H2|T]).
?- countTo(4,[_,X|_]).
?- countTo(2,[H1,H2|T]).
?- countTo(2,[H1,H2,H3|T]).
```

What do you observe? Why? What effect does using the underlining symbol ("_") have?

As defined, the list constructor ".(_, _)" (or the operator "[_|_]" if we use the infix notation) allows us to add an item `E` *at the head* of a list `L` to build a larger list. Any other type of operation on lists must be defined in Prolog.

For example, following the definition of lists given at the beginning of this section and adopting the infix representation, it is easy to inductively define a predicate `islist` that confirms whether a given data is a list or not. This is done by defining a base case for the empty list `[]` and a recursive case for non-empty lists, which are formed using the `[_|_]` constructor:

```
% islist(L), L is a list
islist([]).
islist([_|T]) :- islist(T).
```

Most Prolog systems provide a good collection of predefined predicates for list manipulation. Two of the most common ones are `member` and `append`:

- The predicate `member(E,L)` allows us to check if an element `E` is contained in the list `L`, although this predicate is invertible and can also be used to extract the elements of a list.

  Exercise 3 *Try the following queries by typing ";" after each interpreter response to explore the entire solution space:*

  ```
  ?- member(2,[5,2,3,2]).
  ?- member(X,[5,2,3]).
  ```

  Exercise 4 *Write your own predicate* `mymember` *from the following code (with errors) that you must correct to get the solution.*

  ```
  mymember(E,[E,_]).
  mymember(E,[H|L]) :- mymember(H,L).
  ```

  The definition of the `member` predicate in Prolog is:

  ```
  % member(E,L), E belongs to L
  member(E,[E|_]).
  member(E,[_|L]) :- member(E,L).
  ```

- The `append(L1,L2,L)` predicate allows you to concatenate the lists `L1` and `L2` to form the list `L`. This predicate is predefined in Prolog but you can define it yourself. In the following exercise we will study how to do it.

Exercise 5 *Define a predicate called* `myappend` *that does exactly the same as* `append` *but defined by yourself. The following code is a starting point but there is an error in it that you must find and correct.*

```
myappend([],L,L).
myappend([E|L1],L2,[X|L3]) :- X = E, myappend(L1,L2,L1).
```

(Note that sometimes Prolog gives warnings about "singleton variables"*. This warning means that there is a variable that appears in one place and that is not used in any other part of the rule. The use of use an anonymous variables (represented with an underline) would probably be more appropriate).*

The definition in Prolog of the predicate[1] `append` is as follows. Why does this work?

```
% append(L1,L2,L), the concatenation of L1 and L2 is L
append([],L,L).
append([E|L1],L2,[E|L3]) :- append(L1,L2,L3).
```

Like the `member` predicate, this predicate is invertible and admits multiple uses.

Exercise 6 *Try the following queries by typing ";" after each interpreter response to explore the entire solution space:*

```
?- append([a,Y],[Z,d],[X,b,c,W]).
?- append(L1,L2,[a,b,c,d]).
```

You can define, with the help of the `member` and `append` predicates, a great variety of predicates, as we will see in the following subsections.

2.2 Flattening a list.

We are going to define the `flatten(L,A)` relationship, where `L` is a *list of lists*, as complex in its nesting as we can imagine, and `A` is the list that results from rearranging the elements contained in the lists nested in a single level. For example:

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

[1] However, this definition does not behave well in all cases, since for a target "`?- append([a], a, L).`" the following answer is obtained "`L=[a|a]`", which is not a list, since "a" is not a list either. A way to solve this problem is to verify that, in effect, the second argument of the predicate `append` is always a list. To do this, it suffices to modify the rules that define the predicate `append`:

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

This fact once again shows that in a language without types the programmer is resposible for checking that each used object has the appropriate type (domain).

The version provided here makes use of the predefined predicate `atomic(X)`, which checks whether the object that is linked to the `X` variable is or is not a simple "atomic" object (i.e., a constant symbol) —an atom in the jargon of Prolog—; that is, an integer or a string of characters). It is also required to use the predefined predicate `not`. Here, and in the next sections, we will use `not` with a purely declarative sense, interpreting it as the connective "¬" and without going into the special treatment of negation in Prolog.

```
% flatten(L, A), A is the result of flattening L
flatten([],[]).
flatten([X|L],[X|P]) :- atomic(X), flatten(L,P).
flatten([X|L],P) :- not atomic(X), flatten(X,P_X),
                    flatten(L,P_L), append(P_X,P_L,P).
```

The above solution indicates that the empty list is already flattened (first clause). On the other hand, there are two cases (second and third clause, respectively) to flatten a generic list `[X|L]`:

1. If the element in the head `X` is atomic, it is already flattened; so just put it on top of the flat list and continue to flatten the rest of the list `L` that is being flattened.

2. If the element in the head `X` is not atomic, it will have to be flattened; then the rest `L` is flattened from the original list and, finally, the lists resulting from the previous process are concatenated.

2.3 The last element of a list.

The element `E` at the top of a list `[E|L]` is directly accessible by unification. Nevertheless, to access the last element of a list, a specific predicate `last(L,U)` must be defined, where `U` is the last item in the list `L`. This predicate can be defined as follows:

```
% last(L,U), U is the last element of the list L
last([U],U).
last([_|L],U) :- last(L,U).
```

A declarative reading of this predicate seems clear: `U` is the last element from the list `[_|L]`, if `U` is the last item in the remaining list `L`; when the list contains only one element (i.e. when the list is of the form `[U]`) it is clear that `U` is the last element. On the other hand, the procedural effect of this predicate is to go through all the list, discarding elements, until reaching the last one.

The `last(U,L)` predicate can also be defined much more concisely using the `append` relation:

```
% last(L,U), U is the last element of the list L
last(L,U) :- append(_,[U],L).
```

The idea behind this definition is that the list `L` is the concatenation, to a list that contains all the elements except the last one, of the list `[U]` formed by the last element. The definition of the predicate `append` and the unification algorithm do the rest.

2.4 Sublists operations.

The `append` predicate can also be used to define operations with sublists. If we consider that a prefix of a list is an initial segment of it and that a suffix is a final segment, we can define these relationships with great ease by making use of of the `append` predicate.

```
% prefix(P,L), P is a prefix of the list L
prefix(P,L) :- islist(L), append(P,_,L).

% suffix(P,L), P is a suffix of the list L
suffix(P,L) :- islist(L), append(_,P,L).

% sublist(S,L), S is a sublist of the list L
sublist(S,L) :- prefix(S,L1), suffix(L1,L).
```

The last definition states that `S` is a sublist of `L` if `S` is prefixed with a suffix of `L`.
Some observations:

1. Many times, checking the domains of the arguments of the defined predicates can affect the performance. This is the reason why these checks are often omitted.

2. On the other hand, the indirect call to other predicates in the body of a rule can also affect performance. For this reason, it may be advisable to replace the call with the body of the predicate to which it is called (appropriately renaming the clause variables so that its head coincides with the call). For example, in the definition of the predicate `sublist`, we can replace the call `prefix(S,L1)` by `append(S,_,L1)`. In the same way, `suffix(L1,L)` would be replaced by `append(_,L1,L)`, which leads to the following more efficient definition for the predicate `sublist`:

   ```
   % sublist(S,L), S is a sublist of the list L
   sublist(S,L) :- append(_,L1,L),append(S,_,L1).
   ```

   The transformation process that we have just described is a particular case of a transformation that is known as *unfolding* and is one of the basic techniques used in the field of automatic transformation of logical programs.

2.5 Reversing a list: accumulation parameters and tail recursion.

List reversal can be defined in terms of the `append` predicate:

```
% inverse(L,I), I is the list obtained by reversing L
inverse([],[]).
inverse([H|T],L) :- inverse(T,Z), append(Z,[H],L).
```

This version is very inefficient because it consumes and reconstructs the original list using the `append` predicate, its cost is quadratic with the number of items in the list to be reversed. In order to eliminate calls to `append` and to achieve greater efficiency, it is necessary to use an *accumulation parameter* which, as its name states, is used to store intermediate results. In our case, we store there the list that is being inverted in its different phases of construction.

```
% inverse(L,I), I is the list obtained by reversing L
% using an accumulation parameter.
inverse(L,I) :- inv(L,[],I).

% inv(List,Accumulator,Inverted)
inv([],I,I).
inv([X|L],A,I) :- inv(L,[X|A],I).
```

Note that the reversed list is built in each call to the predicate `inv`, adding an element `X` to the list accumulated in the previous step by using the operator "`[_|_]`", instead of using the `append` predicate. Figure 2 illustrates this process.

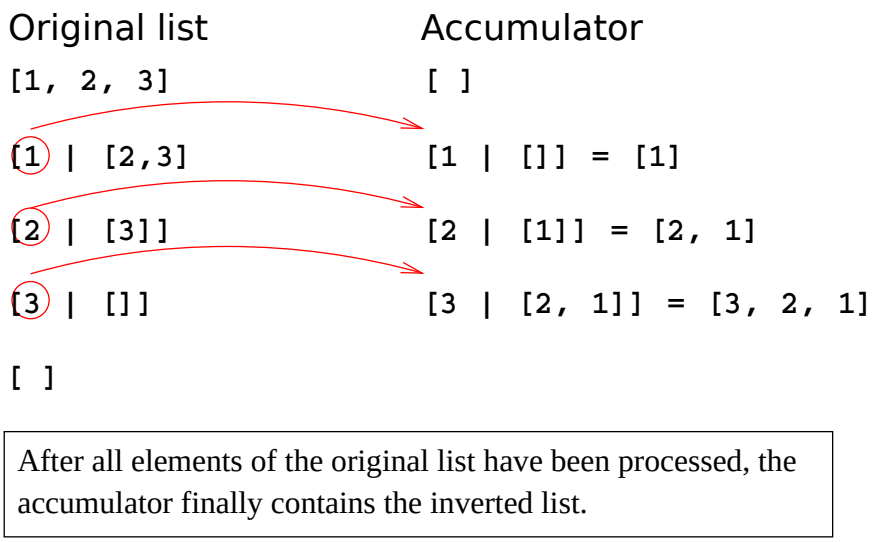

Figure 2: Phases in the inversion of a list using an accumulation parameter.

The technique of accumulation parameters is a general technique that can be applied to many problems (eg, the efficient computation of Fibonacci numbers).

2.6 Final exercises with lists

Exercise 7 *Write a binary predicate* `swap` *that accepts a list and generates a similar list with the first two elements swapped.*

Exercise 8 *What makes the following mysterious predicate? Why?*

```
mistery([],0).
mistery([_|T],N) :- mistery(T,M), N is M+1.
```

Exercise 9 *Complete the following Prolog program to check if a collection of elements is a subset of another one:*

```
subset([],_).
subset([A|X],Y) :- member(A,Y), [          ].
```

$\boxed{A}$ `subset(X,Y)`

$\boxed{B}$ `append(X,[A],Y)`

$\boxed{C}$ `subset(Y,X)`

$\boxed{D}$ `member(X,Y)`

Exercise 10 *Complete the following Prolog program to check if a list is sorted:*

```
sorted([X]).
[          ] :- X =< Y, sorted([Y|Ys]).
```

$\boxed{A}$ `sorted([X|Y,Ys])`

$\boxed{B}$ `sorted([X,[Y|Ys]])`

$\boxed{C}$ `sorted([X,Y|Ys])`

$\boxed{D}$ `sorted(X,Y,Ys)`

Exercise 11 *Complete the following logical program so that, given an integer and a list of integers, delete the occurrences of that integer from the list. The predefined predicates " ==" and "\==" represent equality and inequality, respectively. For example, the call* `delete(3,[1,2,3,1,2,3],L)` *computes the answer* `L = [1,2,1,2]`.

```
remove(_,[],[]).
remove(C,[X|R],L) :- X == C, remove(C,R,L).
remove(C,[X|R],W) :- X \== C, [          ], [          ].
```

$\boxed{A}$ `remove(C,R,L), W = [X|L]`

$\boxed{B}$ `remove(C,R,W), L = [X|W]`

$\boxed{C}$ `remove(C,R,L), W = L`

$\boxed{D}$ `remove(C,[X|R],L), W = L`