Computer Science Journal of Moldova, vol.**3**, no.1(7), 1995

List manipulation in Turbo Prolog

V.Cotelea

Abstract

The present paper is concerned with list processing in Turbo Prolog language. It doesn't claim to be an exhaustive description of operations which can be performed upon lists. Nevertheless adduced programs are most representative, more or less known to specialists in logic programming domain. By means of examples are explained the list manipulation techniques, use of recursion, declarative comparison of predicates, analysis and fixation of acceptable prototypes and some problems of predicates' inconsistency. Index terms. Prolog, list, recursion.

Lists are the basic data structure in Turbo Prolog programs and the most valuable programming technique is recursion. A list is an ordered sequence of arbitrary length of elements. The order of elements in the sequence is essential. In case the order is changed we are having a new list. At the same time the list may be conceived (if desired) at a logical level as a set, going aside the order of elements.

The list in Turbo Prolog is a binary structure,i.e. the first argument is the first element, the second is the recursive defined rest of list. Thus the list is represented as $[X|Xs]$, where arguments X and Xs describe the head and the tail correspondingly. Here and than, when it is possible, if X is the list head, then Xs represents the list tail.

Recursion allows us to process the elements of list. To avoid an infinite recursion there is a necessity of presence of a constant symbol. This is the empty list, which is presented by [ ]. As the list elements can be anything: constants, variables, and surely structures that contain other lists. The only prerequisite is that all elements in a list must belong to the same type.

Let's recollect, that the simplest program in Turbo Prolog language consists of the following compartments [1]: domains, where the data types are defined; predicates, in which predicates are declared; clauses, where facts, clauses, rules are described; and goal compartment.

While processing the lists there may often occur the situation, when clauses which define a predicate may satisfy some goal, while for another goal they meet with failure. Let's examine the following definition of $list(Xs)$ predicate, which has true value, if Xs is a list, the last element of which has as tail the empty list.

$$list([\_|Xs]) :- list(Xs).$$
$$list([\,]).$$

For these rules the goals $list([a,b,c,d])$, $list([\,])$, $list([f(1,2)])$ will get the true values. In case we put the question $list(Xs)$, then the program will enter an infinite loop. The above mentioned predicate has the prototype (i), i.e. input prototype (the term "prototype" is taken from reference [2]). To eliminate that shortage the clauses may be written down in other order:

$$list([\,]).$$
$$list([\_|Xs]) :- list(Xs).$$

But for queries of $list(Xs)$ form, the answer always have to be $Xs = [\,]$, if the goal is in the program and have enter an infinite loop in case the goal is entered from the keyboard. Thus this predicate tests only whether its argument is a list. So the routine has to check right through until it finds a $[\,]$ as tail. Now, consider an another definition of the list predicate, which is not subject to enter an infinite loop.

$$list([\,]).$$
$$list([\_|\_]).$$

This *list* predicate's description has also the output prototype (o). It tests list's head only, but not the fact, that its tail contains empty list. By this description a term is a list if it is an empty list or it has a head and a tail. Thereby the last definition is not a strict test of the list, as

the above, but instead it wouldn't enter an infinite loop in the case the argument is a variable [3]. And here we shall have two solutions for the $list(Xs)$ goal: $Xs = [\,]$ and $Xs = [\_|\_]$, depending on the clause order. Therefore it is recommended, the predicate prototypes to be defined explicitly by means of $free()$ and $bound()$ (if Turbo Prolog does not know the value of variable and knows its value, respectively). The above mentioned examples show the importance of predicate analysis from the point of view of prototypes they express.

The basic problem in list processing is the membership problem. It consists in determination whether a given element is the list head. If so, examination of the list is successful; if not, then the tail of the list is considered to belong to that of the element. So, we study the head, but in this case — the head of the tail etc. If in recursion we reach the empty tail and the process is still not finished, then our search has failed. In Turbo Prolog language this may be rendered as follows:

$$member(X, [X|\_]).$$
$$member(X, [\_|Xs]) :- member(X, Xs).$$

These clauses have several interesting applications. We ascertain among them those, which can be used to test the membership of the given element to the given list by the $member(b, [a, b, c])$ goal; which can be used to discover all list's elements, e.g., $member(X, [a, b, c])$; and which can be used to generate lists with a given thing as member, for example, $member(b, Xs)$. The last goal seems strange, but there exist programs, which are based on such a use of *member* relation.

Let's notify a characteristic feature for query of $member(X, [a, b, c])$ type. If the goal is in program, than as soon as the first element is found, the reception of others is stopped. But using the fact, that *member* is a nondetermined predicate, we can go from the first clause to the second one by $fail$. So the system will determine the following element etc, till the whole list would be exhausted. As for $member(b, Xs)$, the result will be $Xs = [b|\_]$, i.e. Xs will be a list with b head and the tail will be any list. But in the common case the last type of goal leads to an infinite loop.

The beauty of Turbo Prolog lies in the fact, that, often we construct

the clauses for a predicate from one point of view, while they work for another as while. As an example of this, we'll now construct a predicate $append(Xs, Ys, Zs)$ to append lists Xs and Ys to a form Zs.

$$append([\,], Ys, Ys).$$
$$append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs)$$

There are various applications of the *append* relation as well as for the *member* relation. The main interpretation consist in reception of junction of two lists, which can be expressed by $append([a, b, c], [d, e], Xs)$ and, which lead to result $Xs = [a, b, c, d, e]$. For queries of $append(Xs, [c, d], [a, b, c, d])$ and $append([a, b], Ys, [a, b, c, d])$ types, the answer consists in finding of the first part $Xs = [a, b]$ and second part $Ys = [c, d]$ of list, which in junction with the second part $[c, d]$ and with the first part $[a, b]$ result in $[a, b, c, d]$ accordingly. The *append* relation may have the (i, i, i) prototype too. For example, queries $append([a, b], [c], [a, b, c])$ and $append([a, b], [c, d], [e, f])$ have the values $true$ and $false$ respectively. This relation also hold if only Zs list is known. For example, to find which two lists could be appended to form a known list, we could use the goal of the form $append(Xs, Ys, [a, b, c])$. For which Turbo Prolog will find the solution: $Xs = [\,], Ys = [a, b, c]$ and $Xs = [a], Ys = [b, c]$ and $Xs = [a, b], Ys = [c]$ and $Xs = [a, b, c], Ys = [\,]$.

The *append* relation may by used to divide a list into two ones — one on the left side and another on the right side of a given element of the list: $append(Xs, [c|Ys], [a, b, c, d, e])$. Solution is $Xs = [a, b]$, $Ys = [d, e]$.

Its easy to verify, that the *member* relation may be defined by the *append* relation:

$$member(X, Ys) :- append(\_, [X|\_], Ys).$$

This definition affirms, that X is an element of the Ys list, if the Ys list may be divided into two lists, where X is the head of the second list.

The *append* relation can be utilized in the definition of the $adjacent(X, Y, Zs)$ predicate, which is true, if X and Y are adjacent

elements in the Zs list.

$$adjacent(X, Y, Zs) : -append(\_, [X, Y|\_], Zs).$$

We can use it to return successive pairs. An example to illustrate this is the $adjacent(X, Y, [a, b, c])$ goal. We can instantiate one of the first two arguments and get the other one, for example the goals $adjacent(X, b, [a, b, c])$ and $adjacent(a, Y, [a, b, c])$. It may be used also to test whether the two items are adjacent in the given list (the (i, i, i) prototype). It is obvious that if the third argument is free then the *adjacent* predicate enters an infinite loop.

Another easy constructed operation with *append* relation is the last relation.

$$last(X, Xs) : -append(\_, [X], Xs).$$

The above clause defines, that X is the last element of the Xs list, if it is the only element of the second list, which forms the Xs list. The *last* predicates has two prototypes. Namely, it can be used to extracts the last element of a list (the (o, i) prototype), and to test whether the given element is the last one of the given list (the (i, i) prototype).

The repeated application of *append* predicate is available to define the $reverse(Xs, Ys)$ relation, where Ys is the reverse list of the Xs list. As an example there may be the $reverse([a, b, c], [c, b, a])$ goal, for which the *true* value is attributed. The most "naive" procedure consists in recursive reversion of the tail, after that the first element is attached to the end of the reversed tail (majority of examples are taken from [4]).

$reverse([\,], [\,]).$
$reverse([X|Xs], Zs) : -reverse(Xs, Ys), append(Ys, [X], Zs).$

It's clear,that the reverse predicate has two prototypes — (i, i) and (i, o). The list to be reversed should be the first argument.

The *reverse* relation may be defined without addressing to the *append* relation. Let's define an auxiliary $reverse(Xs, Ys, Zs)$ predicate of arity three to which the value *true* is attributed in case Zs is

the junction of the Ys list and the inverted Xs list.

$$reverse(Xs, Ys) :- reverse(Xs, [\,], Ys).$$
$$reverse([X|Xs], Acc, Ys) :- reverse(Xs, [X|Acc], Ys).$$
$$reverse([\,], Ys, Ys).$$

This routine is more efficient than the former one. It's not difficult to examine the derivation trees of both predicates and to observe, that for the first predicate the tree dimension is growing polynomially in respect to reversed list's size, while for the last predicate the dependence is linear. This takes place due to the use of an additional data structure *Acc*.

Let's consider the $sublist(Xs, Ys)$ predicate, which receives the *true* value, if the Xs list is a sublist of the Ys list. We have to mention, that the order of list's elements is essential here. For example, $[b, c]$ is a sublist of $[a, b, c, d]$, while $[c, b]$ is not.

At the beginning we shall examine, for simplicity, two types of sublists, that are described by two predicates — $prefix$ and $suffix$. The inclusion of auxiliary predicates is a well–known and efficient logic programming technique, furthermore, they may represent a separate interest. The $prefix(Xs, Ys)$ predicate holds, if Xs is a beginning sublist of the Ys list, e.g. $prefix([a, b], [a, b, c])$. The antipodal predicate $suffix(Xs, Ys)$ affirms, that Xs is a final sublist of Ys, for example $suffix([b, c], [a, b, c])$.

$$prefix([\,], \_).$$
$$prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).$$

$$suffix(Xs, Xs).$$
$$suffix(Xs, [\_|Ys]) :- suffix(Xs, Ys).$$

Therefore a sublist can be described in terms of prefix and suffix — a sublist of a list is the suffix of the list prefix; or a sublist of a list is the prefix of the list suffix.

a: $sublist(Xs, Ys) :- prefix(Ps, Ys), suffix(Xs, Ps).$
b: $sublist(Xs, Ys) :- prefix(Xs, Ss), suffix(Ss, Ys).$

The first clause define the rule according to which Xs is a sublist of the Ys list, if there exists such a Ps prefix of the Ys list for which Xs is a suffix. By analogy with the first definition, the second one says that Xs is a sublist of the Ys list, if there is such a Ss suffix of the Ys list for which Xs is a prefix.

The *sublist* relation may be also recursively defined only by the $prefix$ relation.

$$sublist(Xs, Ys) : -prefix(Xs, Ys).$$
$$sublist(Xs, [\_|Ys]) : -sublist(Xs, Ys).$$

The first clause (the basic clause) of this definition says that Xs is a sublist of the Ys list, whether Xs is a prefix of Ys. The second clause (the recursive clause) define, that Xs is a sublist of the $[\_|Ys]$ list, if Xs is a sublist of the Ys list.

It is evident that the *member* relation may be considered as a particular case of the *sublist* predicate, if it is defined as follows.

$$member(X, Xs) : -sublist([X], Xs).$$

That is X is an element of the Xs list, whether the $[X]$ list with the only X element is a sublist of the Xs list.

As stated above, the *append* predicate may be used to find the two lists that could be appended to form the given list. It follows that the $prefix$ and $suffix$ predicates can be rendered with the aid of the *append* relation.

$$prefix(Xs, Ys) : -append(Xs, \_, Ys).$$
$$suffix(Xs, Ys) : -append(\_, Xs, Ys).$$

Substituting the $prefix$ and $suffix$ predicates in (a) and (b) by their expressions in the *append* terms we'll obtain two supplementary definitions of the *sublist* predicate:

c: $sublist(Xs, Ys) : -append(Ps, \_, Ys), append(\_, Xs, Ps).$
d: $sublist(Xs, Ys) : -append(Xs, \_, Ss), append(\_, Ss, Ys).$

These relations are neat but inefficient compared to the precedent *sublist* predicates.

The next predicate to be considered expresses a relation between lists and numbers. The $listlen(Xs, N)$ predicate says, that the Xs list's length is N, i.e. the Xs list consists of N elements. For example, the $listlen([a, b], 2)$ goal is true in accordance with the *listlen* predicate's semantics.

$$listlen([\_|Xs], N) : -listlen(Xs, N1), N = N1 + 1.$$
$$listlen([\,], 0).$$

What is the multitude of prototypes for this predicate? One is the (i, i) prototype above described, which tests whether $[a, b]$ is of length 2. The other (i, o) — may be represented by the $listlen([a, b], N)$ goal and is being reduced to estimate the length of the $[a, b]$ list.

A new variant of the predicate under consideration may be written as follows.

$$listlen([\_|Xs], N) : - N > 0, N1 = N - 1, listlen(Xs, N1).$$
$$listlen([\,], 0).$$

This definition can be used either to test the length of a list (the (i, i) prototype) or to generate lists of a given length (the (o, i) prototype). On the other hand, it can't determine the length of a list. For instance, the $listlen([a, b, c], N)$ goal falls short of expectation because it is subject to enter an infinite loop.

The $listindex(Xs, N, X)$ predicate has three possible prototypes (i, i, i), (i, o, i), (i, i, o), i.e. it can verify, if in the Xs list the N–est element is X; it determines on what place in the Xs list is the X element; and determines what element is found on the place N in the Xs list.

$$listindex([X|\_], 1, X) : -!.$$
$$listindex([\_|Xs], N, Y) : -$$
$$\quad bound(N), !, N > 1, M = N - 1, listindex(Xs, M, Y).$$
$$listindex([\_|Xs], N, Y) : -bound(Y), listindex(Xs, M, Y), N = M + 1.$$

As is observed there can be distinguished two types of concepts of predicate definitions: procedural and declarative. The Prolog language is a declarative language, but this does not mean that during

the programming process there are not used at all procedural considerations. Declarative considerations prevail, mainly during the time of solving the problem of definitions' validity. As a matter of fact, in Prolog there are harmoniously combined these two methods: procedural construction of programs and declarative interpretation of predicate definitions. We'll show this during the elaboration of a program which eliminates the elements from a list.

First step consist in definition the semantics of the predicate $delete(X, Xs, Ys)$. For elimination of an element from a list three arguments are necessary: the X element meant to elimination; the Xs list, which can include X; and the Ys list, which does not contain any X element.

We pass to the procedural approach, considering the recursive part of the definition of the *delete* predicate. The recursive argument is $[X|Xs]$. There are two cases: X is the element, that has to be eliminated from the list or is the element that has not to be eliminated from the list. In the first case, we describe the recursive elimination of X from Xs. Such a rule can be the clause $delete(X, [X|Xs], Ys) :- delete(X, Xs, Ys)$. In the second case, where the element, which is verified for elimination differs from X, the clause looks like the first, but the Ys list will have the head different from X, and the tail will examined for excluding X — $delete(X, [Z|Xs], [Z|Ys]) :- X <> Z,\ delete(X, Xs, Ys)$.

Declarative approach in the first case: "Eliminating X from the $[X|Xs]$ list we obtain Ys, if eliminating X from Xs we obtain Ys". Declarative point of view in the second case: "Eliminating X from $[Z|Xs]$ we obtain the $[Z|Ys]$ list, if X differs from Z, and eliminating X from Xs we obtain Ys". The limiting condition is being declared simply: "By elimination of an any element from the empty list, we result the empty list" — $delete(\_, [\,], [\,])$. So the total definition of the delete predicate is as follows:

$$
\begin{array}{l}
delete(X, [X|Xs], Ys) :- delete(X, Xs, Ys).\\
delete(X, [Z|Xs], [Z|Ys]) :- X <> Z, delete(X, Xs, Ys).\\
delete(\_, [\,], [\,]).
\end{array}
$$

Let's make an analysis of this definition. If from the second clause the condition $X <> Z$ is excluded, then another variant of the *delete* relation is obtained. It will be less acceptable, because from the Xs list not always will be eliminated all X entries. E.g., the goals $delete(b, [a, b, c, d], [a, c, d])$, $delete(b, [a, b, c], [a, c, b])$, $delete(b, [a, b, c, d], [a, b, c])$ and $delete(b, [a, b, b], [a, b, c, b])$ all hold for that variant of definition.

There should be mentioned, that the above definition, as well as the new variant accept goal, in which lists do not contain elements that have to be eliminated. For example, $delete(b, [a], [a])$. There exist applications for which such a feature is not desired.

We will examine another predicate which as distinct from the *delete* predicate excludes from a list only one entry of a given element.

$$select(X, [X|Xs], Xs).$$
$$select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).$$

Let's observe that the description of the select predicate is a hybrid of the *member* and *delete* predicates' definitions. The declarative perception of this definition is: "Eliminating the X element from the $[X|Xs]$ list we obtain the Xs list or eliminating X from $[Y|Ys]$ we obtain the $[Y|Zs]$ list, if eliminating X from Xs we obtain the Zs list".

A special interest presents the (i, o, i) prototype of the select relation. For example the $select(b, Ys, [a, c, d])$ goal forms the Ys list introducing the b element into the $[a, c, d]$ list. To emphasize this prototype it is frequently defined a separate relation $insert(X, Ys, Zs)$ which has the (i, i, o) prototype and where Zs is the received list after the insertion of X in the Ys list.

$$insert(x, Ys, Zs) :- select(X, Zs, Ys).$$

The verification of the fact that the elements in a list are arranged in a non–decreasing order may be effected through two clauses. The first rule affirms that any list with a single element is ordered. The second clause says that a list is ordered, whether the first element of

the list is not greater than the second one and the list's tail which begins by the second element is ordered.

$$ordered([\_]).$$
$$ordered([X, Y|Ys]) : -X \Leftarrow Y, ordered([Y|Ys]).$$

The program of partition of a list into two ones resembles the program of elimination of the list's elements. It consist of checking whether the head of the current list is greater or not than the element which determines the partition.

$$partition([X|Xs], Y, [X|Ls], Bs) : -X \Leftarrow Y,$$
$$partition(Xs, Y, Ls, Bs).$$
$$partition([X|Xs], Y, Ls, [X|Bs]) : -X > Y, partition(Xs, Y, Ls, Bs).$$
$$partition([\,], \_, [\,], [\,]).$$

The declarative interpretations of this definition are: "The partition of a list with X and Xs as head and tail respectively in relation to the Y element yields the $[X|Ls]$ and Bs lists, if X is not greater than Y, and the partition of the Xs list in relation to Y yields the Ls and Bs lists" for the first clause. And for the second one it is "The partition of a list with the X head and the Xs tail in relation to Y yields the Ls and $[X|Bs]$ lists, if X is greater than Y and the partition of Xs in relation to Y yields Ls and Bs". For the limiting rule it will be "The empty list is partitioned by any element into two empty lists".

The *partition* predicate may be efficiently used for quick sort of a list. The quick sort conception lies in the choice of a list's element and in the partition of the list into two lists. One contains greater elements than the selected element and another contains the remainder. In the next program the first element of the list is the selected element in comparison to which the partition is made.

$$quicksort([X|Xs], Ys) : -$$
$$partition(Xs, X, Littles, Bigs),$$
$$quicksort(Littles, Ls),$$
$$quicksort(Bigs, Bs),$$
$$append(Ls, [X|Bs], Ys).$$
$$quicksort([\,], [\,]).$$

The recursive clause of the quicksort predicate's definition declares: "Ys is the sorted variant of the $[X|Xs]$ list, whether the $Littles$ and $Bigs$ lists are the result of the Xs list's partition in relation to X; and the Ls and Bs lists are the result of sorting of the $Littles$ and $Bigs$ lists; and Ys is the result of junction of Ls and $[X|Bs]$.

A list may have other lists as members. The procedure of flattening a list involves addition of all elements of such lists to the main list in place of these lists.

$$
\begin{aligned}
&flatten([\,],[\,]) :- !.\\
&flatten([X|Xs],[X|Ys]) :- not(list(X)), flatten(Xs,Ys), !.\\
&flatten([X|Xs],Ys) :- \; flatten(X,FX),\\
&\qquad\qquad\qquad\quad flatten(Xs,FXs),\\
&\qquad\qquad\qquad\quad append(FX,FXs,Ys).
\end{aligned}
$$

This is the simplest variant of the flattening procedure which uses a twofold recursion. The initial constraint statement tells that the flattening of the empty list produces an empty list. The second clause puts in a stack the head of the intended list for flattening it. Here the $not(list(X))$ restriction is necessary in order not to apply the rule in case X is a list. The third clause to flatten an arbitrary $[X|Xs]$ list, where X itself is a list, begins with flattening of the X head and Xs tail and afterwards appends the flattened lists.

Although the meaning of this program is evident, it achieves not the most efficient method of flattening.

Lists are an obvious representation of sets, provided they do not make no assumption about ordering and repetition of lists' elements. Below some predicates will be examined which conceive the lists as sets.

The $members$ relation can be defined easily in terms of $member$.

$$
\begin{aligned}
&members([X|Xs],Ys) :- member(X,Ys), members(Xs,Ys).\\
&members([\,],\_).
\end{aligned}
$$

This predicate determines if every element of Xs is an element of Ys. So it does not deal with the classic notion of subset. For example, $members([b,b],[a,b,c])$ refers the multitude of solutions of the predicate

and so has rather narrow area of application. More of that, if the first or the second argument are variable, then the program cycles. Any argument has to be of the bound type, because of the simple motive, that it is appealed to the *member* relation, described at the beginning of the article. For example, there will be no answer to the $members(Xs, [a, b, c])$ query, because Xs permits the repetition of the elements, and that is why there will be an infinite number of solutions.

These constraints are eliminated by the *selects* relation that deals with the subset in the classic sense of the word.

$$selects([X|Xs], Ys) :- select(X, Ys, Ys1), selects(Xs, Ys1).$$
$$selects([\,], \_).$$

The *selects* relation permits not more repetitions of the elements in the first list than in the second. Because of this particularity the program always ends, if the second argument is of the *bound* type. Goals of the form $selects(Xs, [a, b, c])$ get as solution all subsets of the $[a, b, c]$ set.

The intersection of the two sets is the collection of elements in both of them.

$$intersect([\,], \_, [\,]).$$
$$intersect([X|Xs], Ys, [X|Zs]) :-$$
$$\quad member(X, Ys), !, intersect(Xs, Ys, Zs).$$
$$intersect([\_|Xs], Ys, Zs) :- intersect(Xs, Ys, Zs).$$

This routine may produce the intersection of the two lists, for example, the $intersect([a, b, c], [b, c, d], Zs)$ goal which has the (i, i, o) prototype. To test whether a given Zs set is the intersection of two given sets (as example may be the goal $intersect([a, b, c], [b, c, d], [c, b])$ with the (i, i, i) prototype), we must see if the resulting list is a permutation of the third argument. Thus the correct use of the (i, i, i) prototype of the *intersect* predicate is up to the programmer.

The union of two sets is the collection of elements in either one, or the other.

$union([\,], Ys, Ys).$
$union([X|Xs], Ys, [X|Zs]) :-$
$\quad not(member(X, Ys)), !, union(Xs, Ys, Zs).$
$union([\_|Xs], Ys, Zs) :- union(Xs, Ys, Zs).$

As well as with the *intersect* relation the $(i, i, 0)$ prototype has to be carefully used. One of the solutions can always be the use of the sorted lists.

Finally, we remark that all the predicates described in the article were tested by the system Turbo Prolog version 2.0 and can be freely used without modifications.

References

[1] K.M.Yin, D.Solomon. Using Turbo Prolog. Moscow, MIR, 1993 (Russian)

[2] A.Janson. Turbo–Prolog compact. Moscow, MIR, 1991 (Russian)

[3] W.F.Clocksin, C.S.Mellish. Programming in Prolog. Moscow, MIR, 1987 (Russian)

[4] L.Sterling, E.Shapiro. The art of Prolog. Advanced programming techniques. Moscow, MIR, 1990, (Russian)

V.Cotelea,
Academy of Economic Studies, Moldova
59, Banulescu–Bodoni str., Kishinev,
277001, Moldova

Received 12 January, 1995