# Reverse of a List: A Simple Case Study in Lisp Programming

TUTORIAL MATERIAL FOR THE LISP LECTURE

**Tudor Jebelean, RISC–Linz, Mai 2010**

**Summary.** We present the process of constructing the mathematical model, designing the algorithm, and elaborating few versions of the Lisp program for reversing a list. Although simple, this example demonstrates some important principles of algorithm design and of programming. Moreover, it illustrates the characteristics of the main programming styles: declarative, functional, and imperative.

**The problem.** We consider the relation of a list being the reverse of another:

$$L \sim L'.$$

For instance:

$$\langle\rangle \sim \langle\rangle,$$

$$\langle 1\rangle \sim \langle 1\rangle,$$

$$\langle 1, 2, 3\rangle \sim \langle 3, 2, 1\rangle.$$

We want to solve the problem:

- **given** a list $L$
- **find** a list $L'$
- **such that** $L \sim L'$.

The first step of the analysis must establish the truth of the statement:

*For any list $L$, there exists a unique list $L'$ such that $L \sim L'$.*

For the present example we will consider this statement obvious, however the proof is an interesting logical exercise which also allows a better understanding of the problem and an approach to its solution.

**The function.** As logical consequence of the above statement, we know that there exists a unique function, call it $R$, with the property:

*For any list* $L$, $L \sim R[L]$.

(In this text we use square brackets like in $F[x]$ for denoting function application and predicate application, in contrast to the usual round parantheses like in $F(x)$.)
This function $R$ is the solution to our problem, but now we must *implement* it.

A first step towards implementation is to find properties of the function $R$. For instance:

$$\begin{aligned} R[\langle\rangle] &= \langle\rangle, \\ R[\langle 1\rangle] &= \langle 1\rangle, \\ R[\langle 1,2,3\rangle] &= \langle 3,2,1\rangle. \end{aligned}$$

The first property appears to be interesting because it handles an important special case of lists. The second property can be clearly generalized to:

*For any object* $x$, $R[\langle x\rangle] = \langle x\rangle$.

The same kind of generalization would be possible for the third property too, but here clearly it would be more interesting to express it for lists of *any length*. If we use the symbol $\smile$ for denoting *concatentation* of two lists, then we can express properties like:

$$\begin{aligned} R[L \smile L'] &= R[L'] \smile R[L], \\ R[\langle x\rangle \smile L] &= R[L] \smile \langle x\rangle, \\ R[L \smile \langle x\rangle] &= \langle x\rangle \smile R[L], \end{aligned}$$

*for any object* $x$ *and for any lists* $L, L'$. (It is easy to check these properties on simple examples.)
Note that such properties often include *universally quantified variables* (like $L$, $x$), which sometimes have a certain type (like $L$ is a *list*). In the sequel we will use $x$ as a universal quantified variable for objects of any type, and $L, L'$ for arbitrary lists. (The statements where these symbols occur are implicitely universally quantified.)

**A declarative program.** Let us pick-up two particular properties:

$$R[\langle\rangle] = \langle\rangle \tag{1}$$

$$R[\langle x\rangle \smile L] = R[L] \smile \langle x\rangle \tag{2}$$

Note that these two properties allow, in a relatively easy way, to compute the value of $R$ for any input. For instance:

$$\begin{aligned} R[\langle 1,2,3\rangle] &= \\ R[\langle 2,3\rangle] \smile \langle 1\rangle &= \\ (R[\langle 3\rangle] \smile \langle 2\rangle) \smile \langle 1\rangle &= \\ ((R[\langle\rangle] \smile \langle 3\rangle) \smile \langle 2\rangle) \smile \langle 1\rangle &= \\ ((\langle\rangle \smile \langle 3\rangle) \smile \langle 2\rangle) \smile \langle 1\rangle &= \\ &\langle 3,2,1\rangle. \end{aligned}$$

This computation is "easy" becase, at each step, there is only one equality among (1), (2) which is applicable from left to right on [a parto of] the current expression to be computed. Therefore, this is already an program, which can be used for computation in an environment that is able to identify and apply the appropriate equalities. Such an environment is called a "rewriting environment" (as for instance *Mathematica* and *Maude*) and the equalities, interpreted as "rewrite rules" form a so called "rewrite based program". The rewrite based programming is an example **declarative programming**, because one *declares* certain properties of the function which is implemented. (Another example of declarative programming is *logic programming*, which allows arbitrary predicates, not only equalities, as for instance *Prolog*.) Rewrite based programs consist in equalities (sometimes conditioned – see below) which have on the left-hand side a term of the form $R$[pattern] (where $R$ does not occur in the pattern) and on the right-hand side an arbitrary expression, possibly containing $R$.

The properties (1), (2) allow to compute the reverse of *any* list. This is so because:

- any list is either of the form $\langle\rangle$ or of the form $\langle x\rangle \smile L$,
- at each step, the recursive call to $R$ applies to a shorter list, thus this process will eventually terminate.

These two properties are a consequence of the fact that the domain of finite lists can be *inductively defined* as follows:

- $\langle\rangle$ is a list,
- if $L$ is a list, then $\langle x\rangle \smile L$ is also a list;
- these are all lists.

(Note that there are also other ways to define the domain of lists inductively.) Thus we can infer a scheme for constructing a rewrite program for a function $F$ on lists, using this inductive definition:

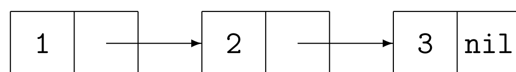
- Find out an expression without $F$ which equals $F[\langle\rangle]$.
- Find out an expression which equals $F[\langle x\rangle \smile L]$ and which may contain $F$, but only as $F[L]$.

This scheme expresses, in fact, a general principle which applies for any function and any inductive defined domain. However, it is not allways possible to construct efficient and effective implementations starting from any inductive definition.

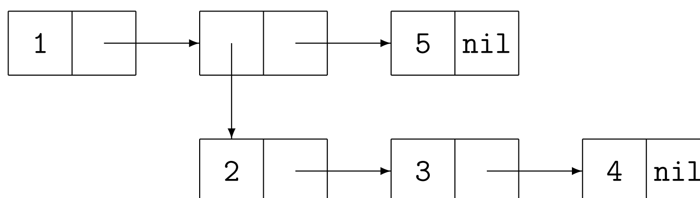
The inductive principle used in the above definition of lists is also used in the programming language *Lisp* and in its evaluation environment. Namely, the Lisp language uses the following notations:

$$\begin{array}{lcl} \langle\rangle & : & (), \\ \langle 1,2,3\rangle & : & \texttt{(1 2 3)}, \\ \langle x\rangle \smile L & : & \texttt{(cons x L)}. \end{array}$$

(In Lisp, function application – like $F[x, z]$ – is denoted by a paranthesized list containing the function name followed by the arguments – like `(F x y)`.) Note that Lisp uses a special function – `cons` – for the operation of prepending an object at the beginning of a list. Moreover, the internal representation of Lisp objects is based on the same inductive principle. Namely, the empty list is represented by a special value named `nil`, while a list is represented by a pointer to a the so called *cons cell.* A `cons` cell contains two elements: the first element of the list and the rest of the list (which is either `nil` or a pointer to a `cons` cell). For instance, the list $\langle 1, 2, 3\rangle$ or `(1 2 3)` is represented internally like this:




Any object occurring in a list can also be a list,
for instance $\langle 1, \langle 2, 3, 4\rangle, 5\rangle$ or `(1 (2 3 4) 5)` is represented as:




Clearly in this representation the function `cons` is very easy to evaluate by a Lisp interpreter or compiler.

**A functional program.** Let us use two list specific functions which decompose a list in its first element (head – $H$) and the rest of its elements (tail – $T$). For instance:

$$H[\langle 1, 2, 3\rangle] = 1,$$
$$T[\langle 1, 2, 3\rangle] = \langle 2, 3\rangle,$$

but $H[\langle\rangle]$ and $T[\langle\rangle]$ are not defined. Using these functions, we can transform the formulae (1) and (2) into the *conditional equalities*:

$$\begin{array}{lcl} L = \langle\rangle & \Rightarrow & R[L] = \langle\rangle, \\ L \neq \langle\rangle & \Rightarrow & R[L] = R[T[L]] \smile \langle H[L]\rangle, \end{array}$$

which are sometimes abbreviated as:

$$\begin{array}{rll} R[L] = \langle\rangle & \textbf{if} & L = \langle\rangle, \\ R[L] = R[T[L]] \smile \langle H[L]\rangle & \textbf{if} & L \neq \langle\rangle. \end{array}$$

Moreover, the two of them can be also expressed together as:

$$R[L] = \left\{ \begin{array}{ll} \langle\rangle & \textbf{if } L = \langle\rangle \\ R[T[L]] \smile \langle H[L]\rangle & \textbf{otherwise}, \end{array} \right.$$

which is the usual notation from mathematical texts (remember the definition of the absolute value of a real number). In some programming languages one uses the syntactically equivalent **if–then–else** construct:

$$R[L] = (\textbf{if } L = \langle\rangle \textbf{ then } \langle\rangle \textbf{ else } R[T[L]] \smile \langle H[L]\rangle).$$

This programming style is called **functional programming** because it describes how to construct *functions* by composition of other functions. The shape of the conditional equalities is now more restricted: on the left-hand side one only allows expressions of the form $F$[variable]. This kind of programs is even easier to interpret, because the environment does not recognizes certain patterns, but only applies functions to arguments.

Lisp is a typical functional programming environment. Lisp provides the functions $H[L]$ and $T[L]$ and it should be obvious that the internal representation of lists described above makes these two functions very easy to compute. In Lisp $H[L]$ is written as `(car L)` and $T[L]$ as `(cdr L)`. The program above is coded as follows:

```
(defun R (L)
   (if (eq L ())
      ()
      (append (R (cdr L)) (list (car L)))))
```

This code is just a syntactic version of the functional definition given before in a more logical style. The reader can check the correspondence by using the additional Lisp language conventions:

| | | |
|---|---|---|
| (`defun` $F$ $(x)$ body) | : | $\underset{x}{\forall}\ F[x] = \text{body}$ |
| (`if` condition $\text{body}_1$ $\text{body}_2$) | : | **if** condition **then** $\text{body}_1$ **else** $\text{body}_2$ |

and the remarks that `eq` is the Lisp equality predicate, the function `append` is concatenation of lists, and the function `list` creates the list of its arguments. Due to the representation of Lisp lists as `cons` cells, concatenation of lists (in particular the appending of an object at the end of a list) is more difficult than `cons`, because it requires the scanning of the first argument. Therefore, the Lisp program above has *quadratic time complexity* (the computing time is proportional to the square of the length of the input list).

**Tail recursion.** If we inspect carefully the evaluation of $R[\langle a,b,c\rangle]$ above, we see that certain computations can be performed earlier, in order to avoid the growth of the intermediate terms. Namely, the evaluation can also be written as:

$$\begin{aligned} R[\langle a,b,c\rangle] \;=\; R[\langle a,b,c\rangle] &\smile \langle\rangle \;= \\ R[\langle b,c\rangle] &\smile \langle a\rangle \;= \\ R[\langle c\rangle] &\smile \langle b,a\rangle \;= \\ R[\langle\rangle] &\smile \langle c,b,a\rangle \;=\; \langle c,b,a\rangle. \end{aligned}$$

This computation can be performed using a new (auxiliary) function $A$ with two arguments:

$$R[\langle a,b,c\rangle] \;=\; A[\langle a,b,c\rangle \quad , \quad \langle\rangle] \;=$$

$$\begin{aligned} A[\langle b,c\rangle \quad &, \quad \langle a\rangle] \; = \\ A[\langle c\rangle \quad &, \quad \langle b,a\rangle] \; = \\ A[\langle\rangle \quad &, \quad \langle c,b,a\rangle] \; = \; \langle c,b,a\rangle, \end{aligned}$$

Obviously we may generalize this to:

$$R[L] = A[L, \langle\rangle]$$

but what is the implementation of $A$? Using the instances above and the inductive definition of lists on the *first* argument, we discover the declarative program:

$$A[\langle\rangle,\ L'] \;=\; L', \tag{3}$$

$$A[\langle x\rangle \smile L,\ L'] \;=\; A[L,\ \langle x\rangle \smile L'], \tag{4}$$

as well as the functional program:

$$\begin{aligned} L = \langle\rangle \quad &\Rightarrow \quad A[L,\ L'] = L', \\ L \neq \langle\rangle \quad &\Rightarrow \quad A[L,\ L'] = A[T[L],\ \langle H[L]\rangle \smile L'], \end{aligned}$$

in equivalent notation:

$$\begin{aligned} A[L,\ L'] = L' \quad &\textbf{if} \quad L = \langle\rangle, \\ A[L,\ L'] = A[T[L],\ \langle H[L]\rangle \smile L'] \quad &\textbf{if} \quad L \neq \langle\rangle. \end{aligned}$$

or:

$$A[L,\ L'] = \begin{cases} L' & \textbf{if } L = \langle\rangle \\ A[T[L],\ \langle H[L]\rangle \smile L'] & \textbf{otherwise}, \end{cases}$$

or:

$$A[L,\ L'] = (\textbf{if } L = \langle\rangle \textbf{ then } L' \textbf{ else } A[T[L],\ \langle H[L]\rangle \smile L']).$$

The function $A$ is called *tail recursive* because the recursive application of $A$ is the *last* ("at tail") operation which is performed on the right-hand side of the equality. This means that the terms which appear during the evaluation (see the example) do not grow, as they do when evaluation the [non-tail-recursive] function $R$. In terms of concrete computer implementation, the evaluation of a non-tail-recursive program needs an arbitrary large amount of memory (usually a stack) for storing the current intermediate term, while a tail-recursive program needs a fixed amount of memory for storing the current arguments of the function (two in our example).

In Lisp this implementation has the following syntax:

```
(defun R (L)
   (A L ()))

(defun A (L Lp)
   (if (eq L ())
      Lp
      (A (cdr L) (cons (car L) Lp))))
```

Moreover, since `A` does not use `append`, but `cons`, this implementation is significantly more efficient: it has *linear time complexity*.

**An imperative program.** The functional tail-recursive program implementing $A$ can be transformed into:

$$
\begin{array}{lll}
R[L] & \{ & L_1 := L; \\
 & & L_2 := \langle\rangle; \\
 & & \mathbf{While}(L_1 \neq \langle\rangle) \quad \{ \quad L_2 := \langle H[L_1]\rangle \smile L_2; \\
 & & \qquad\qquad\qquad\qquad\quad L_1 := T[L_1]; \\
 & & \qquad\qquad\qquad\quad \}; \\
 & & \mathbf{Return}[L_2]; \\
 & \} &
\end{array}
$$

This is an **imperative program**, because it specifies the *imperative* commands that are to be executed by the computer in order to compute the value of the function. We use above an ad-hoc syntax, similar to the one of most imperative programming languages:

- "$F[X]\{\text{body}\}$": the implementation of a function $F$ of argument $X$.
- "$v$ := expression": store the value of the expression under the name $v$. (Note that ":=" is different from the logical equality "=".)
- "**While**(condition){body}": repeat body as long as the condition holds.
- "**Return**(expression)": exit the program returning the value of the expression as the value of the function which is implemented.

The imperative program is quite far from the logical properties of the original function $R$, however it is much more easier to compile or to intrepret.

In C or in Java the program may look like this:

```
reverse(list L) {
  list L1, L2;
  L1 = L;
  L2 = emptylist();
  while(nonempty(L1)){
    L2 = cons(head(L1), L2);
    L1 = tail(L1);
  };
  return Lp;
}
```

In Lisp we can also write such an imperative program, using the *iterative* construct `do`:

```
(defun R (L)
  (do ((L2 () (cons (car L1) L2))
       (L1 L (cdr L1)))
      ((eq L1 ())
       L2)))
```

The body of the function starts with a list of triples: each triple declares an internal variable, its initial value, and the operation which must be executed at each step in order to update this variable. Next comes the condition for terminating the loop, and then the return value. The previous tail recursive version of $R$ can be transformed automatically into this iterative version, even using a Lisp program!

**Conclusion.** Any programmer with a minimal training is able to write the imperative program for the simple problem of reversing a list, *without* having to go explicitely through the process described above. For programmers with more training, this is also possible for much more complicated problems. However, as the problems are more complex, the incidence of errors or even failures to design the program increases. This is why it is useful to study the algorithm design principles demonstrated above and to apply them explicitely in our programming practice. As we see from this example, the most important part of programming is not the actual writing of the code in the respective programming language, but the construction of the mathematical model and the design of the algorithm such that it is correct and efficient.