Chapter 2
Lists Everywhere

2.1 Introduction

As we saw in Chap. 1, Lisp is an acronym for **LIS**t **P**rocessing, that is, all the data and programming in Lisp is based on an organizational structure characteristic of this programming language called "list". This chapter is entirely dedicated to this pivotal aspect of Lisp and we shall dedicate space enough for showing how to create, manage, modify, reset and eliminate lists.

The previous chapter gave you a quick introduction to simple numerical calculations using Lisp as a way of getting a first touch with the language. Every Lisp expression exposed was genuinely Lisp, but all the exposed material was even far from scratching the surface of it. Now it is time to ask you for a bit of concentration in order to undertake the new material. The concepts you are going to find in the sections of this chapter will give you a solid understanding of the conceptual pillars of Lisp. In fact, when you feel you have understood entirely all the ideas exposed in this chapter you will start to get comfortable with the language, being able to appreciate its philosophy, elegance, and flexibility.

As usual, we shall use a practical approach, getting help in this case from the use of examples taken from meteorology, geometry, art and discrete event simulation. There is a lot of material in this chapter and sometimes you will find that the information supplied is a bit dense. Be patient and take the time you need. As Italian people usually say, "*piano piano si va lontano*", that is, "*going slowly you'll reach distant places*". This is not a literal translation, of course, but gives you an idea about the attitude you should take while reading this chapter.

L. Argüelles Méndez, *A Practical Introduction to Fuzzy Logic using LISP*,
Studies in Fuzziness and Soft Computing 327,
DOI 10.1007/978-3-319-23186-0_2

2.2 Atoms and Lists

Berlin has a relatively cold climate. Summer is something between mild and warm, with cold winters, while the rest of the months are more chilly than mild. In any case, the *Berliner Luft* (Berlin air) is famous, and Berliners are really proud of it. This paragraph, almost a small meteorology report, serves us for introducing the following Lisp expression:

(-3 -2 1 4 8 11 13 12 9 6 2 -1)

This is a list formed by the monthly average of low temperatures in Berlin, from data obtained from Climatedata (2014). Another example of list could be:

(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

This list contains the months of the year. A list is anything enclosed between an open and a closed parenthesis. If an element from a list can not be expressed in a simpler way, that element is called an **atom**. An example of atom is "-3", as is also "Jan". Generalizing we can say:

A list is a formal collection of elements enclosed by parenthesis whose elements are either atoms or lists

However, a list can not live isolated from the Lisp environment. If we supply a list to the prompt in a Lisp session we obtain the following:

> **(-3 -2 1 4 8 11 13 12 9 6 2 -1)**
: *ERR: illegal parameter type : -2*

This error message means that Lisp does not know what to do with a solitary list. Lisp needs that some activity must be performed on lists, or better said, the *grammar* of Lisp uses a structure where solitary lists are not allowed. Here, what we call structure can be as simple as a function that takes a list as an argument or as complex as an entire program written in the language. For better fixing ideas we should remember again that the acronym for Lisp is LISt Processing, and a list always needs some processing. When we add some structure, things start to work perfectly well:

> **(setq Berlin-lows '(-3 -2 1 4 8 11 13 12 9 6 2 -1))**
: *(-3 -2 1 4 8 11 13 12 9 6 2 -1)*

As we already know, this is a well-formed Lisp expression because, among other things, it contains a matched number of parentheses. The *setq* function is also known from the previous chapter, and it serves to assign values to a variable. At least this is what we have learnt so far. In the above expression it seems clear that something is assigned to the just created variable "Berlin-lows". Typing it we can observe that NewLisp returns:

> Berlin-lows
: *(-3 -2 1 4 8 11 13 12 9 6 2 -1)*

Now the symbol *Berlin-lows* stores the list *(-3 -2 1 4 8 11 13 12 9 6 2 -1)*, but an important detail remains unexplained and it is the quote operator that is usually represented in Lisp by the quote sign ‘. This important operator prevents Lisp from evaluating lisp expressions located at its right. In order to know how many expressions are not evaluated, we must distinguish two cases, depending on the lisp expression located immediately after the quote operator:

- If an atom is found just after the quote operator, Lisp stops its evaluation processing until finding a new atom or list. That is, only the quoted atom is not evaluated.
- If an open parenthesis is found just after the quote operator, Lisp stops its evaluation processing until finding its matching right parenthesis.

We shall understand it better with several examples:

> ‘(+ 1 2)
: *(+ 1 2)*

Without the quote, Lisp evaluates the list immediately, as you already know:

> (+ 1 2)
: *3*

By the way, alternatively you can also type:

> (quote (+ 1 2))
: *(+ 1 2)*

> (setq a-nested-list ‘(a b (p1 (x1 y1) p2 (x2 y2))))
: *(a b (p1 (x1 y1) p2 (x2 y2)))*

The *quote* operator in Lisp has a powerful counterpart function named *(eval)* that is imperative to introduce in this moment. Using it forces Lisp to evaluate the expression(s) located at its right. Let us type the following expression:

> (setq operation ‘(+ 7 3))
: (+ 7 3)

Calling *(eval)* in this moment we shall have:

> (eval operation)
: 10

This function is extremely powerful, and although purist Lisp programmers tended to consider its use as an abuse, it certainly helps to save code in certain occasions. We shall have the opportunity to observe how does it work in future chapters of this book.

Again at our list representing the monthly average of low temperatures in Berlin, the expression:

(setq Berlin-lows '(-3 -2 1 4 8 11 13 12 9 6 2 -1))

processes the list of temperatures by means of the adequate grammatical use of setq and quote. As a result the list is bound to the symbol Berlin-lows, converting it into a variable. The distinction between symbol and variable is important and we are going to discuss it immediately.

The list introduced some paragraphs above:

(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

is a list composed by the symbols Jan, Feb, … etc. These symbols are not variables, since they are not still bound to any value. Lisp itself helps us to identify symbols, lists and atoms by some useful functions called *Predicates*. Predicates are special functions that always return either *true* or *nil*, the equivalent of "false" in Lisp. The name of every predicate always ends with a question mark. Let us type the following Lisp expressions with attention:

> (atom? a)
: true

Since "a", either containing a value or not, is indivisible, is clearly an atom.

> (symbol? 'a)
: true

Here "a" is quoted, that is, it is not evaluated by Lisp, so it is a completely legal symbol for Lisp. It does not matter if it holds any associated value.

> a
: nil

The direct evaluation of "a" returns nil because it does not contains anything. We have not yet assigned any value to it by means of the function *setq* so Lisp evaluates it to nil. Now let us assign a value to "a", numerical in this case:

> (setq a 10)
: 10

if now we evaluate it:

> a
: 10

it is bounded to the value 10, and not only that: after being bounded to a numerical value "a" is a variable of numeric type. But not only is a variable, it continues being a symbol:

> (symbol? ‘a)
: true

Note again that in this expression we have quoted “a”. If we evaluate again it without quoting:

> (symbol? a)
: nil

Lisp immediately evaluates “a”. Since it is now a variable, it stores the numerical value 10 so this expression is equivalent to:

> (symbol? 10)
: nil

This is absolutely correct since “10” is a numerical value, not a symbol. We can follow using the number? predicate:

> (number? a)
: true

and as you can easily imagine:

> (number? ‘a)
: nil

Speaking about predicates, The language itself provides a function in order to know if something is a list:

> (list? ‘(-3 -2 1 4 8 11 13 12 9 6 2 -1))
: true

> (list? Berlin-lows)
: *true*

> (symbol? ‘Berlin-lows)
: *true*

As you can see, this is a wonderful computing game although maybe a bit confusing at first. Do not worry if these concepts are a bit difficult to grasp in this moment. As they say, practice makes perfect and my best advice is to enjoy several Lisp sessions where you create lists, symbols and, by assigning values, variables. Incidentally, I suspect it is harder if you already know other computer languages as C++ or Java, languages that are strongly typed. As we wrote in Chap. 1, Lisp automatically takes care of memory allocation and management so you can concentrate in program design and thus, being more productive. Anyway, for C programmers we could give a hint: symbols in Lisp are in fact pointers in disguise. The good thing is that Lisp manages both pointers (memory addresses) and values in a transparent way for the Lisp user.

After these theoretical concepts, and as a small relaxation, we can seize the opportunity to introduce a practical function embedded in NewLisp for calculating

Table 2.1 Calculating basic statistics with the NewLisp built-in function *(stats)*. See text

Statistical element	Calculated values for Berlin-lows
Number of values	12
Mean of values	5
Average deviation from mean value	4.833333333
Standard deviation	5.640760748
Variance	31.81818182
Skew	0
Kurtosis	-1.661427211

simple statistics. The name of this function is *stats*, and we can apply it to the list of low temperatures in Berlin for observing how does it work:

```
> (stats Berlin-lows)
: (12 5 4.833333333 5.640760748 31.81818182 0 -1.661427211)
```

The function *(stats)* takes as argument a list containing a set of numerical values and after some almost instantaneous calculations returns statistical values ordered in the following sequence: Number of values, mean of values, average deviation from mean value, standard deviation, variance and skew and kurtosis of distribution, as shown in Table 2.1. The actual version of NewLisp (v.10.5.4 as of this writing) contains several built-in statistical and probabilistic functions that are useful in science and engineering. Since the mission of this book is to allow the reader to understand the theories of fuzzy-sets and fuzzy-logic we shall not dedicate more space to calculate statistics, but the reader should be aware that these NewLisp built-in functions deserve to be explored. The NewLisp Manual and Reference provide all the needed information to start doing statistics with this Lisp implementation.

For calculating the mean of values with the knowledge learnt from Chap. 1 we would have used, for example the following expression:

```
> (setq average-min (div (add -3 -2 1 4 8 11 13 12 9 6 2 -1) 12))
: 5
```

As promised, Lisp offers us a way of making things in a very powerful way. And we are only starting with it.

2.3 First and Rest

The first element in a list has an enormous importance in Lisp since its combination of symbol and value determines if it is a list of data or a list with a call to a function. In fact, when Lisp finds a list, it immediately evaluates its first element in order to know what to do with it. For example, the first element in the list of low monthly

temperatures for Berlin (-3 -2 1 4 8 11 13 12 9 6 2 -1) is "-3", so Lisp identifies it as a list where no function is called. On the other hand, the first element of the list (add -3 -2 1 4 8 11 13 12 9 6 2 -1) is "*add*" and since it is included in the list of symbols of Lisp that represent functions then treats the rest of the list as arguments that must be provided to the function call. Having into account this grammatical rule of the language, it is not a surprise that Lisp incorporates two functions for accessing the first element of a list and the rest of elements. In the NewLisp implementation, these functions are named *first* and *rest*, respectively (in Common Lisp and many other Lisp implementations these functions are named *car* and *cdr* as an homage to their first use on the IBM704 computer). Its use is more than intuitive:

> (first '(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))
: Jan

Please note that the list of months is quoted. Let us observe how the function *rest* does it work:

> (rest '(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))
: (Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

The natural question now is: Ok, it is easy to understand the use of the functions *first* and *rest*, in fact it is one of the more easy concepts exposed so far in the book, but what kind of things can we do with them aside obtaining the first element and the rest of elements in a list? Important answer: Using *(first)* and *(rest)* we can access any element on any list. In order to type less code, let us begin typing the following in a Lisp session:

> (setq months '(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))
: (Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

> (first months)
: Jan

> (rest months)
: (Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

You should already know why the variable *months* is not quoted in the above two expressions, but the important thing is that the first element from the list returned after the evaluation of *(rest months)* corresponds to the second month, February. Let us rewrite the previous expression in order to store the list it returns on the variable *rest-1*:

> (setq rest-1 (rest months))
: (Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

Now, let us continue typing:

> (setq second-month (first rest-1))
: Feb

> (setq rest-2 (rest rest-1))
: (Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

> (setq third-month (first rest-2))
: Mar

Reiterating this procedure by using several calls to *(first rest-i)* for *i* going from 1 to 11 we can access any month from the list from February to December. For January, in order to generalize the algorithm just exposed, we could have written:

> (setq rest-0 months)
: (Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

Now rest-0 contains the full list of months. Obtaining the first element in the sequence of lists *rest-0*, *rest-1*, …, *rest-11*, we can access any month in the year. This type of procedure for traversing lists is typical of Lisp and we shall use them frequently. Yes, at this stage it is a lot of typing, but our agreement for learning Lisp and ultimately fuzzy logic at the beginning of the book requires typing Lisp code. In the next chapter we shall learn to write functions and that will allow us to go from interactive Lisp sessions to Lisp programming. For now, the typing will continue. However, and almost without noticing it, you are learning Lisp at a good pace.

2.4 Building Lists

Without doubt, *(first)* and *(rest)* are powerful functions, but accessing elements from a list is so frequent in Lisp programming that Lisp designers soon realized that a special function should be created for accessing any element from a list without using *first* and *rest*. The name of this wanted function is *n*th. Having into account the previous expression:

(setq months '(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))

Now we can write:

> (nth 0 months)
: Jan

> (nth 5 months)
: Jun

> (nth 11 months)
: Dec

Please note the special indexing for the function *n*th: the first element in a list is acceded by the number 0, not by the number 1, so the last element in a list of n elements can be acceded by using n − 1 as the index for *n*th. By the way, in the

same manner there is a function for accessing the first member of a list, there is also another one for directly accessing its last element:

> (last months)
: Dec

When using *first*, *last* and especially *n*th, it is usual, if not mandatory before making a call to *n*th, to use the Lisp function *length.* As its name suggests, it returns the number of elements in a list:

> (length months)
: 12

then, another way to access the last element of a list could be:

> (nth (- (length months) 1) months)
: Dec

Nobody does it that way, of course, but it is a safe way to avoid errors while programming. If we write:

> (nth 12 months)
*: ERR: invalid list index in function n*th

This error would break the execution of any Lisp program, so as a general rule:

Always use the function *length* before using *n*th and remember that indexing in lists starts at zero

Figure 2.1 helps to shed more light into the relationships between *(first)*, *(rest)*, *(last)* and *(n*th*)*:

The trick to understand it all is as simple as observing the first block, where we do *list* = $rest_0$. Understanding $rest_0$ as the whole list help us to get used to zero indexing for the function *n*th. This is exactly what we did with the expression *(setq rest-0 months)* some paragraphs above.

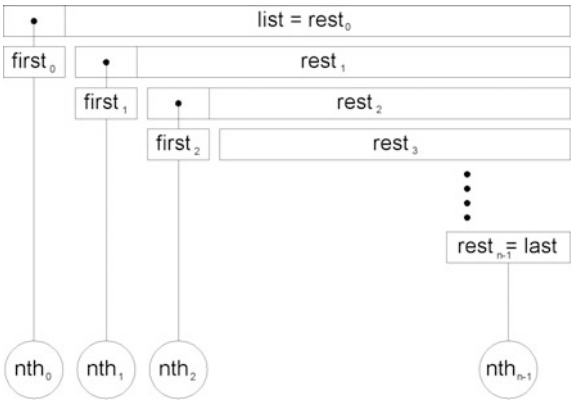

Fig. 2.1 A graphical representation of the relationships between the Lisp functions *(first)*, *(rest)* and *(nth)*

After arriving to this point, we are ready to learn how to build lists. Until now we have used the assignment function *setq* for linking a list to a symbol, and the symbols *Berlin-lows* and *months* were built this way. From these two symbols we are going to build a more complex list where every month and its correspondent low temperature will be paired in a sublist. Let us type the following:

> (setq mt1 (cons (nth 0 months) (nth 0 Berlin-lows)))
: (Jan -3)

Here the new function is *(cons)*. It takes two arguments, either atoms or lists, and then constructs a new list with them. Let us see other examples:

> (cons (first months) (rest months))
: (Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

> (cons (first months) (last months))
: (Jan Dec)

But take care with the arguments. If we type *(cons Jan Feb)* Lisp returns the list *(nil nil)*. Why? Because both *Jan* and *Feb* are not yet bounded to any value and consequently Lisp evaluate these symbols to *nil*. It is a different story if we type *(cons 'Jan 'Feb)*, then Lisp returns the list *(Jan Feb)*.

We have chosen *mt1* as the name of the symbol representing January and its temperature almost like an acronym for **m**onth and **t**emperature, appending a figure one for linking it to the number of the month. It is always a good strategy to choose the name of symbols and variables with a clear meaning for us as Lisp programmers. We shall revisit this idea in the next chapter. For now let us continue for the rest of months:

> (setq mt2 (cons (nth 1 months) (nth 1 Berlin-lows)))
: (Feb -2)

> (setq mt3 (cons (nth 2 months) (nth 2 Berlin-lows)))
: (Mar 1)
…
…

and after some heavy typing (not at all, you can copy, past and modify the code), we reach December:

> (setq mt12 (cons (nth 11 months) (nth 11 Berlin-lows)))
: (Dec -1)

And now, another new function, named *list*, makes the magic:

> (setq months-temps (list mt1 mt2 mt3 mt4 mt5 mt6 mt7 mt8 mt9 mt10 mt11 mt12))
: ((Jan -3) (Feb -2) (Mar 1) (Apr 4) (May 8) (Jun 11) (Jul 13) (Aug 12) (Sep 9) (Oct 6) (Nov 2) (Dec -1))

Now, every element of the list represented by the symbol *months-temps* is in itself a list of two elements. The *list* function takes any number of arguments (12 arguments have been used in this example) and then, every argument is evaluated and then is used as an element for building a new list. After creating the list *months-temps* we can type, for example:

> (nth 5 months-temps)
: (Jun 11)

But this is not the end of the story. This is the perfect moment for introducing another function named *assoc*. Let us see how does it works with an example:

> (assoc 'Aug months-temps)
: (Aug 12)

If you suspect we are entering the realms of database programming you are completely right. The function *assoc* takes the first argument passed to the function and then uses it as a key for searching the list passed as the second argument. If a match is found, then it returns the member list. If no match is found, it returns nil. See now how easy is to access any data from the list *months-temps*:

> (first (assoc 'May months-temps))
: May

> (last (assoc 'May months-temps))
: 8

This function not only serves for accessing elements in a list, it also permits to directly change information in lists using *setq*. Let us see an example for changing the minimum temperature in July from data from, for example, Canary Islands:

> (setq (assoc 'Jul months-temps) '(Jul 21))
: (Jul 21)

after *setq*, *assoc* uses *Jul* as the key symbol for finding the sublist *(Jul temperature)* and then substitutes it by the sublist given in the second argument, in this case, (*Jul 21*). Now we can observe the changes made:

> (nth 6 months-temps)
: (Jul 21)

We could have written, for example:

> (setq (assoc 'Jul months-temps) '(Jul 21 Canary-Islands))
: (Jul 21 Canary-Islands)

and now, the contents for the *July* sublist are, obviously:

> (nth 6 months-temps)
: (Jul 21 Canary-Islands)

Using *assoc* in this way has a destructive effect on the original list, in this case, on *months-temps*. While some Lisp functions preserve the original list (*n*th or *length*, for example, come quickly to mind) other ones modify the original data. In many cases it is a safe strategy to make first a copy of the original data and then manipulate the copy. For now, we can restore the original data for minimum temperature in Berlin in July simply writing:

> (setq (assoc 'Jul months-temps) '(Jul 13))
: (Jul 13)

2.5 Some Geometry and then Some Art, Too

In general, lists of the type:

$$((atom_{11}\ atom_{12})\ (atom_{21}\ atom_{22})\ \ldots\ (atom_{n1}\ atom_{n2}))$$

are extremely useful in Lisp, since they can be an excellent representation of many observational data and phenomena in science, engineering or technical fields. Needless to say, it is the natural way in Lisp for representing pairs of geometric coordinates, giving access to the wonderful world of graphics. Since fuzzy-logic can be seen, as we shall see in the second part of this book, as a matter of geometry, we must seize the opportunity to play a bit with geometric forms in this moment.

As a general rule, we can represent rectangles, trapeziums and squares by means of four coordinates, triangles by means of three coordinates and circles by means of a coordinate for the centre and another number for its radius. Table 2.2 shows some list structures for representing geometrical shapes in a flat, two dimensions space:

If the shape is a rectangle or a square, its list representation is even simpler by using the coordinates of the two points of one of its diagonals. Let us type some values in order to create some rectangles:

> (setq pt1 '(5 -4) pt2 '(8 12)) (setq R1 (list pt1 pt2))
: (8 12)
: ((5 -4) (8 12))

The rectangle identified by the symbol R1 now storages two points, defined by its coordinates. By the way, we have written several Lisp expressions in the same

Table 2.2 List representation of simple 2D geometrical forms

Geometric form	Lisp coordinate representation
Rectangle, trapezium, square	$((x_1\ y_1)\ (x_2\ y_2)\ (x_3\ y_3)\ (x_4\ y_4))$
Rectangle, square (by diagonal)	$((x_1\ y_1)\ (x_2\ y_2))$
Triangle	$((x_1\ y_1)\ (x_2\ y_2)\ (x_3\ y_3)$
Circle	$((x_1\ y_1)\ r)$

line after the prompt in order to save some space. Now let us repeat the operation for some other rectangles:

> (setq pt3 '(0 0) pt4 '(10 10)) (setq R2 (list pt3 pt4))
: (10 10)
: ((0 0) (10 10))

> (setq pt5 '(2 -1) pt6 '(16 2)) (setq R3 (list pt5 pt6))
: (16 2)
: ((2 -1) (16 2))

> (setq pt7 '(6.5 1) pt8 '(13.5 8)) (setq R4 (list pt7 pt8))
: (13.5 8)
: ((6.5 1) (13.5 8))

After having well defined rectangles R1, R2, R3 and R4, now we can build a geometrical composition with them named "*Mondrian-style*" as an homage to Piet Mondrian, the famous Dutch painter:

> (setq Mondrian-style (list R1 R2 R3 R4))
: (((5 -4) (8 12)) ((0 0) (10 10)) ((2 -1) (16 2)) ((6.5 1) (13.5 8)))

Figure 2.2 is a visual representation of the list *Mondrian-style*.

Pablo Picasso was a bit more complex than Mondrian. Aside rectangles we must include now circles and triangles following the structure depicted in Table 2.2. Let us start with a rectangle:

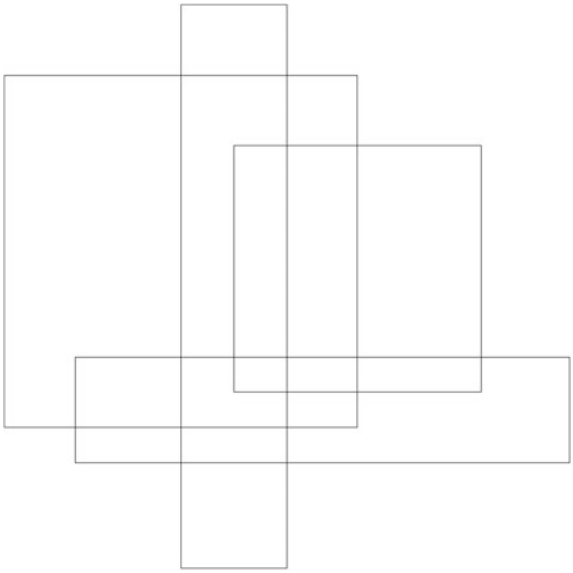

Fig. 2.2 Four *rectangles* representing a composition inspired in the style of Piet Mondrian

> (setq pt9 '(5 -4) pt10 '(8 12)) (setq R5 (list pt9 pt10))
: (8 12)
: ((5 -4) (8 12))

Now it is the time for circles C1 and C2. We shall define them in only one line of Lisp code:

> (setq C1 '((8 2.5) 1.5) C2 '((6 9) 0.5))
: ((6 9) 0.5)

And now it is time for the triangles:

> (setq T1 '((6 9) (10 6.75) (6 4.5)))
: ((6 9) (10 6.75) (6 4.5))

> (setq T2 '((1.438 2.5) (7.5 -1) (7.5 6)))
: ((1.438 2.5) (7.5 -1) (7.5 6))

Finally let us join the shapes in order to build the composition:

> (setq Picasso-style (list R5 C1 C2 T1 T2))
: (((5 -4) (8 12)) ((8 2.5) 1.5) ((6 9) 0.5) ((6 9) (10 6.75) (6 4.5)) ((1.438 2.5) (7.5 -1)
(7.5 6)))

In Fig. 2.3 we can see now our personal, Lisp based Picasso composition.

We have forgotten an important detail: The name of the paintings! For this we shall use a new function named "*append*" that, as its name implies, appends data to the tail of an existing list. Let us use it:

> (setq Mondrian-style (append Mondrian-style '("Composition #968")))
: (((5 -4) (8 12)) ((0 0) (10 10)) ((2 -1) (16 2)) ((6.5 1) (13.5 8)) "Composition #968")

> (setq Picasso-style (append Picasso-style '("Pregnant woman")))
: (((5 -4) (8 12)) ((8 2.5) 1.5) ((6 9) 0.5) ((6 9) (10 6.75) (6 4.5)) ((1.438 2.5)
(7.5 -1) (7.5 6)) "Pregnant woman")

This seems a bit complex at first sight, because aside the function *(append)* we have introduced at the same time a new type of data named "*string*". Let us start with the function *append*. It appends anything to an existing list, but first the new thing(s) to append must be enclosed into a list and second, the function *append* is not destructive, meaning that the original list remains unchanged. Let us see this important detail with a simpler example:

> (setq my-list '(a b c))
: (a b c)
> (append my-list '(d))
: (a b c d)

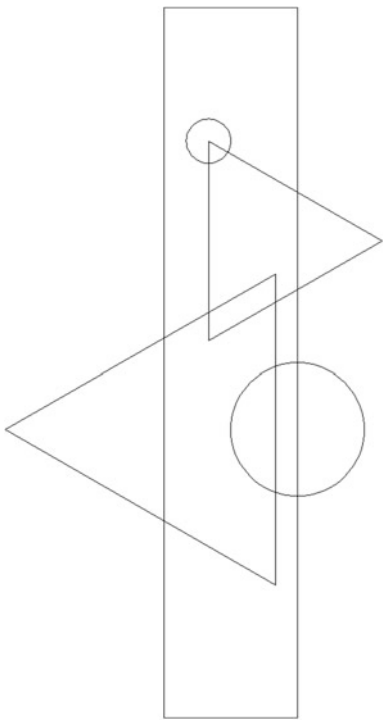

Fig. 2.3 A *rectangle*, two *circles* and two *triangles* for representing a Picasso style composition

but if we evaluate what is now pointed by the symbol *my-list,* a surprises appears:

> my-list
: (a b c)

After applying *append* to a list it returns another list with an appended new element at its tail, but it does not even touch the original list. If we want the original list to be modified we need to use *setq* as follows: *(setq my-list (append my-list '(d)))* After introducing this Lisp expression at the prompt then *my-list* contains (a b c d), as desired. This is the explanation for using *setq* in the expressions:

(setq Mondrian-style (append Mondrian-style '("Composition #968")))
(setq Picasso-style (append Picasso-style '("Pregnant woman")))

"Composition #968" and "Pregnant woman" are strings, that is, a type of data composed by characters. Interestingly, strings are not symbols:

> (symbol? (last Mondrian-style))
: nil

We shall dedicate more time to strings in a future chapter. This type of date is important because when we write Lisp programs that ask an user for data, either

numerical or alphanumerical, Lisp reads the input (usually from the keyboard) and returns strings.

Returning to our peculiar paintings, yes, I am hearing you about the issue of having the name of the paintings at the end of the lists. Usually, a painting is identified by its name, so it seems natural to have it located at the first position. Well, this is a very easy to solve problem by means of using the function *(reverse)*. As you have imagined, it reverses the order of elements in a list:

> (reverse Mondrian-style)
: ("Composition #968" ((6.5 1) (13.5 8)) ((2 -1) (16 2)) ((0 0) (10 10)) ((5 -4) (8 12)))

(reverse Picasso-style)
: ("Pregnant woman" ((1.438 2.5) (7.5 -1) (7.5 6)) ((6 9) (10 6.75) (6 4.5)) ((6 9) 0.5)
((8 2.5) 1.5) ((5 -4) (8 12)))

This function is destructive, modifying the original list. However the use of *reverse* over *reverse* restores the list at its original status, that is: *(reverse (reverse a-list))* → *a-list*

Now it is time to finally make a small gallery, a collection of two paintings:

> (setq my-gallery (list Mondrian-style Picasso-style))
: (("Composition #968" ((6.5 1) (13.5 8)) ((2 -1) (16 2)) ((0 0) (10 10)) ((5 -4) (8 12))) ("Pregnant woman" ((1.438 2.5) (7.5 -1) (7.5 6)) ((6 9) (10 6.75) (6 4.5)) ((6 9) 0.5) ((8 2.5) 1.5) ((5 -4) (8 12))))

The representation of my-gallery can be shown in Fig. 2.4.

Well, the representation of Fig. 2.4 is not exact, as any attentive reader has already noticed. First, there is a common rectangle in the two compositions and second, coordinates from the two compositions share a common space in order to

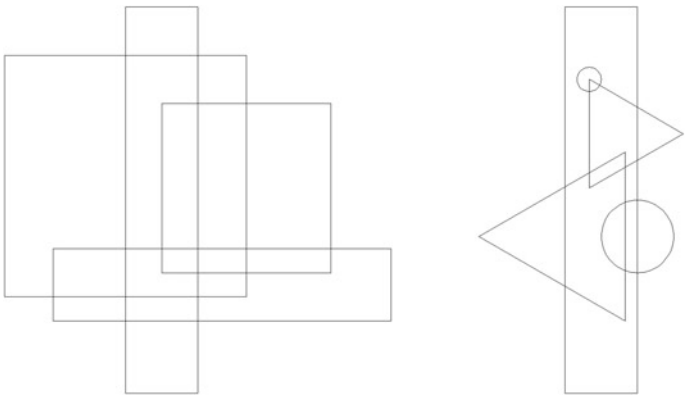

Fig. 2.4 The my-gallery collection, representing Mondrian and Picasso look-alikes

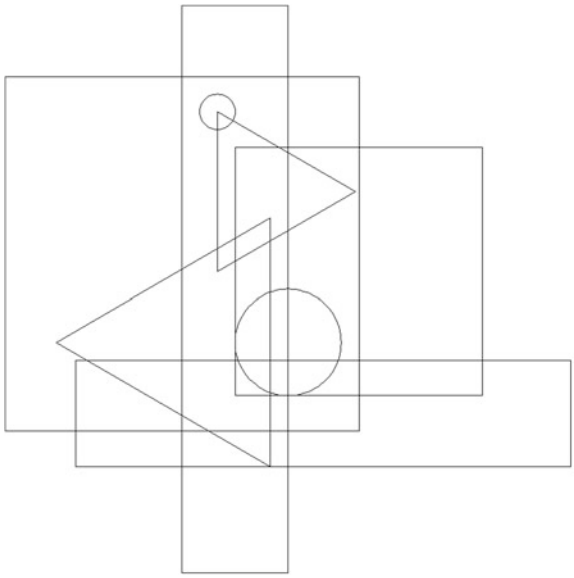

Fig. 2.5 The my-gallery collection, true representation

use small and simple coordinates in the Lisp expressions. The true representation of *my-gallery* is shown in Fig. 2.5 that seems to suggest that Mondrian and Picasso compositions maybe do not mix well.

A positive shift for every x value in the coordinates for the *Picasso-style* list should have been applied for obtaining the image shown in Fig. 2.4. We shall learn how to perform such a shift in the next chapter.

In Table 2.2 we have seen how to represent some geometric shapes in 2D. If we assume that the geometric forms are resting over a horizontal axis, it is even easier to represent them. As we can see in Table 2.3, in this case a rectangle only need two values of x for representing the location of its base and a height, h. A square is even simpler because given the base by x_1 and x_2 then its height is equal to the distance between x_1 and x_2. The base of a triangle comes represented by its extremes at x_1 x_3 and then its third point is given by the point x_2,h. Finally, a trapezium is represented by the extremes at its base x_1 and x_4 and the points x_2,h_1 and x_3,h_2.

Let us see a simple example of a triangle and two trapeziums resting over a horizontal axis using some lisp expressions:

> (setq trapezium-1 '(2 (4 4) (6 4) 8))
: (2 (4 4) (6 4) 8)

> (setq triangle '(6 (8 4) 10))
: (6 (8 4) 10)

> (setq trapezium-2 '(8 (10 4) (12 4) 14))
: (8 (10 4) (12 4) 14)

Table 2.3 List representation of simple 2D geometrical forms when resting over a horizontal axis

Geometric form	Lisp coordinate representation
Rectangle, square	(x_1 x_2) h)
Square	(x_1 x_2)
Triangle	(x_1 (x_2 h) x_3)
Trapezium	(x_1 (x_2 h_1) (x_3 h_2) x_4)

as we know, for joining these geometric forms into a single Lisp expression, we can type:

(setq simple-composition (list trapezium-1 triangle trapezium-2))
: ((2 (4 4) (6 4) 8) (6 (8 4) 10) (8 (10 4) (12 4) 14))

Figure 2.6 shows the graphic representation of the just created symbol *simple-composition*. This type of geometry seems simple at first, but as we shall see in this book, has a big conceptual importance.

Histograms and other related graphics are also based on using both vertical and horizontal axes where values can be absolute or relative. For example, our list of minimum temperatures by month in Berlin can be shown easily on a two-dimensional chart. As previously seen in Sect. 2.2, the symbol *Berlin-lows* points to the list *(-3 -2 1 4 8 11 13 12 9 6 2 -1)*. The range of stored temperatures goes from -3 to 13, resulting into a whole range of 16°. Since there are twelve values corresponding to all the months in the year, we can choose a space between months of two units for the horizontal axis, yielding a 2D region of 24 × 16 units for representing the data. Manipulating the list as we saw in Sect. 2.4 by means of the functions *cons* and *list*, or directly by *setq*, we can easily arrive at the following list of points $(x_i\ y_i)$ where x_i are the abscissa values and y_i the temperature (ordinate) values:

((0 -3) (2 -2) (4 1) (6 4) (8 8) (10 11) (12 13) (14 12) (16 9) (18 6) (20 2) (22 -1))

Joining the points stored in this list with lines we obtain a representation of the minimum temperatures in Berlin, as shown in Fig. 2.7.

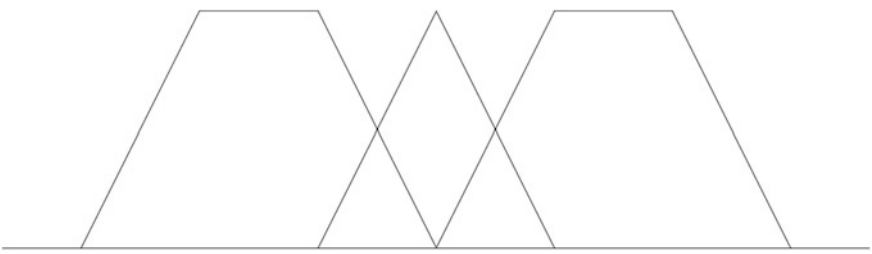

Fig. 2.6 Graphic representation of the list *simple-composition*

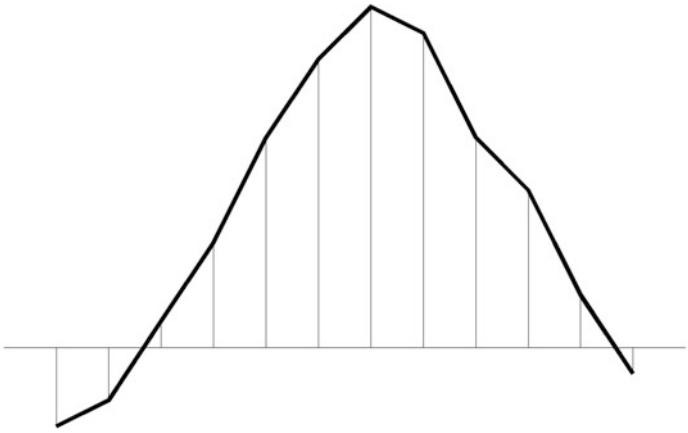

Fig. 2.7 Minimum temperatures in Berlin by month

Going from two dimensions to three dimensions (3D) graphics implies an additional degree of complexity. Some simple 3D geometric forms are shown as lists in Table 2.4.

One of the causes of the bigger complexity of 3D objects is that its orientation in space must be defined. Hence, for example, the eight points needed for representing a truncated pyramid. However, if a 3D object rests or hovers over a flat horizontal surface, some simplification results, especially for parallelepipeds. In this situation, a parallelepiped can be expressed in list form by:

$$((x_1\ y_1)\ (x_2\ y_2)\ (x_3\ y_3)\ (x_4\ y_4)\ h_1\ h_2)$$

where h_1 and h_2 are, respectively, the free height over the plane XY (floor), and the intrinsic height of the parallelepiped. This geometrical element is one of the most important ones in architecture. Figure 2.8 shows the Barcelona Pavilion (1929) by the famous architect Mies Van der Rohe. Practically all the components of the Pavilion could be expressed by means of Lisp expressions as the one shown above.

Before ending this section on geometry and Lisp lists, it deserves to be commented that one of the most popular Computer Aided Design (CAD) applications in the industry, AutoCAD, uses Lisp as an embedded programming language. Named

Table 2.4 List representation of simple 3D geometrical forms

Geometric form	Lisp coordinate representation
Parallelepiped	$((x_1\ y_1)\ (x_2\ y_2)\ (x_3\ y_3)\ (x_4\ y_4)\ (x_5\ y_5)\ (x_6\ y_6))$
Sphere	$((x_1\ y_1)\ (x_2\ y_2)\ (x_3\ y_3)\ r)$
Pyramid	$((x_1\ y_1)\ (x_2\ y_2)\ (x_3\ y_3)\ (x_4\ y_4)\ h)$
Truncated pyramid	$((x_1\ y_1)\ (x_2\ y_2)\ (x_3\ y_3)\ (x_4\ y_4)\ (x_5\ y_5)\ (x_6\ y_6)\ (x_7\ y_7)\ (x_8\ y_8))$

Fig. 2.8 Barcelona Pavilion, 1929. Architect: Ludwig Mies van der Rohe *Photograph* by the author

"AutoLISP", it is a close Lisp dialect to NewLisp, so, after reading this book you would be in a good position to use Lisp in a 3D graphic environment. And not only that, you could apply fuzzy logic in Computer Aided Design and graphics, too. Sounds exciting? I can assure you it really is.

2.6 A World Full of Queues

Everyday we experience queues in our lives. Before paying in the supermarket, just before entering inside a plane at the airport or while being trapped in a traffic jam in the highway, just to name a few examples. Queue theory is important in factories, in production management, data processing, financial strategies and so on. Lisp has two functions, named *pop* and *push* that are especially suited to manage and model queues. They were not created as a direct solution to problems derived from queue theory but for list management, but since this is a practical book we are going to learn to use them while modeling a toll station in a highway.

As it is well known, a toll station is basically a facility located on a fixed point in a highway where cars arrive, are serviced by a human being or a machine (the driver gets a ticket and/or pays a variable sum of money) and after some service time, they leave the queue. Let us say we are interested in two things: first, to record all the cars that enter the facility, their car plate and the enter and exit time from the toll station. This is very useful to the organism, either public or private, that

manages the highway and it also serves for supplying information in special cases, let us say, as an example, for a police requirement. Second, we are interested in modeling the facility itself, that is, to represent how many cars are waiting in the toll station in real time. Before entering into the details, let us take a quick view at how *pop* and *push* do work. Let us start creating an empty list:

> (setq queue '())
: ()

Now three elements, a, b and c, enter in the list sequentially:

> (push 'a queue)
: (a)

> (push 'b queue)
: (b a)

> (push 'c queue)
: (c b a)

As it can be seen, the *push* function adds elements into a list "pushing" them sequentially from the frontal part of the list to its back. Under this point of view, it is a list builder, but also a function that modifies the original contents of the list just after being applied. Note also that the list is built in reversal order. This is an important detail. Let us use now the function pop. It does not need any argument:

> (pop queue)
: c

Let us now observe what remains in *queue* after using *pop*:

> queue
: (b a)

In other words, while the function *push* introduces elements into a list, the use of *pop* returns the first element of a list and at the same time eliminates it from the list. As already suggested, it is useful for queue representation, but only for queues where its discipline is Last-In-First-Out, or LIFO, as it is usually named in queue theory. This is not what we were waiting for to apply to our model of toll station whose queue discipline is in fact First-In-First-Out, or FIFO, that is, the first car that enters the toll station is the first car that leaves it. Sadly, Lisp does not provide a built-in function for managing elements in a list in a FIFO way, so we shall need to do some list manipulation management in order to perform the desired behavior in the facility.

To better fix ideas let us assume that some devices are already installed in the toll station: a camera that reads the car plates from cars by means of an artificial vision system and another sophisticated device composed by a chronometer and a set of photoelectric cells for registering the arrival and departure time for every car. We also assume that the facility starts its morning shift at 8:00. Table 2.5 shows some car plate data and arrival and departure times for several cars being served at the toll station.

Table 2.5 Example data for cars entering a toll-station in a highway

Car plate	Arrival time	Departure time
CKT8623	08:02:12	08:02:54
GWG2719	08:02:18	08:04:32
MKA8772	08:02:25	08:05:55
DYN2140	13:15:21	13:16:22

For simplicity, we shall represent the arrival and departure times as simple numbers, i.e., 08:02:12 will be represented by 80212, 13:15:21 by 131521, and so on. Our desired working model is that every car enters the queue from the "left" of the list and departs at its "right". First let us create the queue and then add some cars to it.

> (setq queue '())
: ()

> (push 'CKT8623 queue)
: (CKT8623)

> (push 'GWG2719 queue)
: (GWG2719 CKT8623)

> (push 'MKA8772 queue)
: (MKA8772 GWG2719 CKT8623)

Between 08:02:12 and 08:02:54 we have three cars in the queue. At 08:02:54 the first car departs, so it must leave the list. The actual contents of *queue* are:

> queue
: (MKA8772 GWG2719 CKT8623)

Now the car that must leave the queue must be the one with the car plate CKT8623, so a call to the function *pop* is not possible. In order to perform a correct working model of the queue we must proceed with caution. The first step is to use the function *reverse*:

> (reverse queue)
: (CKT8623 GWG2719 MKA8772)

Now we can apply the function *pop*:

> (pop queue)
: CKT8623

> queue
: (GWG2719 MKA8772)

We are almost done. A second call to the function *reverse* will finish the algorithm for representing a FIFO queue in Lisp:

> (reverse queue)
: (MKA8772 GWG2719)

Reiterating the procedure, car MKA8772 would leave the queue at 08:05:55. At this moment, the list *queue* would be equal to the empty list, *()* until the car DYN2140 arrives at 13:16:22.

Meanwhile, and at the same time, we must record all the cars that enter the facility. This is easier to do:

> (setq toll '())
: ()

> (push '(CKT8623 80212 80254) toll)
: ((CKT8623 80212 80254))

> (push '(GWG2719 80218 80432) toll)
: ((GWG2719 80218 80432) (CKT8623 80212 80254))

> (push '(MKA8772 80225 80555) toll)
: ((MKA8772 80225 80555) (GWG2719 80218 80432) (CKT8623 80212 80254))

> (push '(DYN2140 131521 131622) toll)
: ((DYN2140 131521 131622) (MKA8772 80225 80555) (GWG2719 80218 80432) (CKT8623 80212 80254))

The list *toll* does not need any treatment with the function *reverse*, because it is used to register all the vehicles served at the facility, but at the end of the day, it can help to make the recorded data easier to read:

> (reverse toll)
: ((CKT8623 80212 80254) (GWG2719 80218 80432) (MKA8772 80225 80555) (DYN2140 131521 131622))

Now let us assume the police tries to track a car with car plate MKA8772 and calls the highway company in order to know if that car has used the facility. At the company, they only need to type:

> (assoc 'MKA8772 toll)
: (MKA8772 80225 80555)

For confirming the police that a car with that plate has been observed at toll station X from 08:02:25 to 08:05:55. A call to all the toll stations in the country would result in a map with the route of the car. By the way, NewLisp incorporates a function named *now* that can be very useful when we try to record time in an application. Let us observe the results of a call to *now* just as I'm writing:

> (now)
: (2014 2 26 13 34 40 592460 56 3 -60 0)

Table 2.6 Structure of the numerical information returned by the NewLisp function *(now)*

Element	Example	Observations
Year	2014	From the Gregorian calendar
Month	2	From 1 to 12
Day	26	From 1 to 31
Hour	13	From 0 to 23, (UT)
Minute	34	From 0 to 59
Second	40	From 0 to 59
Microsecond	592,460	From 0 to 999,999
Day of current year	56	Begins at 1 for January, 1st
Day of current week	3	From 1 to 7. Starts at Monday
Time zone offset (min)	-60	West of Greenwich meridian
Daylight saving time	0	From 0 to 6

The meaning of the elements of the list returned by a call to the function *(now)* is shown in Table 2.6.

With the use of *(now), (pop), (push) and (random)* (we shall introduce this function in the next section) and some other programming resources, Lisp can be used for solving Discrete Event Simulation problems. See for example Banks et al. (2009), Sturgul (1999). Until recently, queue studies have been studied mainly by simulation, but relatively recent works seem to suggest that Fuzzy-Logic represents a good approach to this kind of problems, too (Zhang 2005).

2.7 Rotate, Extend and Flat

For finishing this chapter we shall introduce yet three NewLisp functions for managing lists. They are not, comparatively, so nuclear to Lisp as the previously ones, that is, they are not present in all the Lisp dialects, but they sure have an additional interest.

For explaining the function *(rotate)*, we should visualize a list as a circular structure where the first and last element would be contiguous elements. For a better and complete understanding of this structure there is not a better example than a French roulette wheel of unique zero located into a casino. Its list representation is as follows:

> (setq roulette '(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36))
: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36)

The function *rotate* uses an integer *n* as parameter and it moves every element for the list *n* positions towards its right if *n* is positive or towards left if *n* is negative. For example:

> (rotate roulette 3)
: (34 35 36 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33)

As can be easily seen every element has moved three positions towards right, and the elements 34, 35 and 36 have rotated from their original positions at the tail of the list to the first ones. Typing *(rotate roulette -3)* would restore the original position of the roulette.

While we are speaking about roulettes, casinos and chance, it seems convenient to introduce the function *random*. This function simply returns a real number between 0 and 1 at random (strictly speaking no computer is able to generate pure random numbers, but the use of pseudo-random numbers is usually enough for representing random events in the real world). Let us see a simple call to random:

> (random)
: 0.283314746

In order to obtain a random integer number from 0 to 36 we should type:

> (setq alpha (integer (mul (random) 37)))
: 24

Now, we can give a spin with a value *alpha* to our roulette and obtain the first element after the rotation of the roulette is applied:

> (first (rotate roulette alpha))
: 13

Now you can argue that a simple call to *(mul (random) 37)* could produce the same roulette simulation, and you are true, but our model not only gets a random number between 0 and 36 but it also models the position of the roulette wheel:

> roulette
: (13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 0 1 2 3 4 5 6 7 8 9 10 11 12)

Another application for *rotate* could be to save a programming step in our example of toll station. As we have seen, after adding some cars to *queue* it pointed to the list *(MKA8772 GWG2719 CKT8623)*. Now if we apply (*rotate queue 1)*, it turns to *(CKT8623 MKA8772 GWG2719)* and therefore, the car with car-plate CKT8623 can exit from the queue graciously by typing *(pop queue)*. As we can see, by using *(rotate)* we have avoided a double use of the function *(reverse)*.

The functions *(extend)* and *(flat)* are simple in their working, but help to make the life of a Lisp user easier. The first one takes several lists as arguments and returns a single list formed by all the elements from the supplied lists. Let's type as an example:

> (setq first-semester '(Jan Feb Mar Apr May Jun))
: (Jan Feb Mar Apr May Jun)

> (setq second-semester '(Jul Aug Sep Oct Nov Dec))
: (Jul Aug Sep Oct Nov Dec)

(setq year '())
: ()

> (extend year first-semester second-semester)
: (Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)

The function *(flat)* analyzes the list supplied as its unique argument and then eliminates any sub-list included in it, returning a list where all the elements are atoms. As an example, remembering Sect. 2.4, we saw that the symbol *months-temps* pointed to the list:

((Jan -3) (Feb -2) (Mar 1) (Apr 4) (May 8) (Jun 11) (Jul 13) (Aug 12) (Sep 9) (Oct 6) (Nov 2) (Dec -1)))

Now, typing:

> (flat month-temps)
: (Jan -3 Feb -2 Mar 1 Apr 4 May 8 Jun 11 Jul 13 Aug 12 Sep 9 Oct 6 Nov 2 Dec -1)

The function *flat* is not destructive, so it lefts intact the function supplied as argument. If we wish that *month-temps* adopts the new, flattened structure, we should type:

> (setq months-temps (flat months-temps))
: (Jan -3 Feb -2 Mar 1 Apr 4 May 8 Jun 11 Jul 13 Aug 12 Sep 9 Oct 6 Nov 2 Dec -1)

This function is especially useful when we are creating a list by a repetitive use of the function *cons*. As we saw in Sect. 2.4, *cons* takes two arguments, but successive calls to *cons* causes a deepening of the formed list. As an example let us use *cons* by building a queue of cars at the toll facility:

> (setq queue (cons 'CKT8623 'GWG2719))
: (CKT8623 GWG2719)

> (setq queue (cons queue 'MKA8772))
: ((CKT8623 GWG2719) MKA8772)

> (setq queue (cons queue 'DYN2140))
: (((CKT8623 GWG2719) MKA8772) DYN2140)

Do you appreciate the problem with the function *(cons)* when used this way? A call to flat solves it all:

> (setq queue (flat queue))
: (CKT8623 GWG2719 MKA8772 DYN2140)

By the way, have you noticed that the combination of the functions *(cons)* and *(flat)* are still another way to model a queue with FIFO discipline? Multiple solutions to a problem are usual when using programming languages, but it is especially true with Lisp due to its high flexibility, as you are starting to appreciate.

In the Sect. 2.1 of this chapter we have learnt how to create lists. Now, at the end of it, we should learn how to reset a list and how to eliminate it from memory. Let us imagine the list *toll* contains now thousands of elements, that is, cars that have entered the facility along the day. In order to reset it, we only need to type:

> (setq toll '())
: ()

Thus becoming an empty list. If we wish to completely eliminate the list *toll* from memory, it is even easier:

> (setq toll nil)
: nil

If what you want is to eliminate a list from memory, it is not necessary, of course, to reset it previously. In any case Lisp takes care for handling these memory issues automatically.

2.8 As a Summary

All the nuclear functions of Lisp, and yet some more, needed for developing FuzzyLisp have been exposed in this chapter, and what is even more important: you are now into a position where you are able to understand the basics of the language. We have made an extensive use of examples, using monthly temperatures in Berlin, abstract art compositions and some examples of queues from the real world in order to progressively introduce the most important functions of Lisp. If you have followed our advice we know it has been a lot of typing at the keyboard, and if you have experimented with examples created by your own imagination, the typing has been certainly impressive. Many typing or conceptual errors have happened you while cruised along this chapter. If this has been so, congratulations because you are now more than ready to explore the next step in being fluent in Lisp: Writing you own functions. We shall dedicate the entire next chapter to functions defined by the Lisp user. Now, a last reminder to all the functions visited in this section of the book will help you to have a good view of your position in the travel of learning Lisp:

- The function (*quote*) or its more often used *quote sign*, prevents Lisp from evaluating Lisp expressions located at its right. If an atom is found just after the quote operator, Lisp stops its evaluation processing until finding a new atom or list. If an open parenthesis is found just after the quote operator, Lisp stops its evaluation processing until finding its matching right parenthesis. Example: *(quote a)* or '$a \rightarrow a$.

- The *quote* operator in Lisp has a powerful counterpart function named *(eval).* It forces Lisp to evaluate the expression(s) located at its right. As an example, *(setq expression '(first (this is great))) → (first (this is great)),* and then *(eval expression) → this.*
- The function *(atom?)* is a predicate that returns *true* if its argument is an atom. Otherwise it returns *nil*, the Lisp equivalent to false. Predicates always end its name with a question mark and only return either *true* or *nil*. As an example *(atom? bird) → true*, but *(atom? '(bird fish))* The *quote* operator in Lisp has a powerful counterpart function named *(eval)* that is imperative to introduce in this moment. Using it forces Lisp to evaluate the expression(s) located at its right. *il*.
- The function *(symbol?)* is another predicate that returns *true* if the supplied argument is a symbol. Otherwise it returns *nil*. Example: *(symbol? 'bird) → true*, but note that *(symbol? bird) → nil*. The use of the quote operator is essential in Lisp.
- The function *(number?)* is just another predicate that returns *true* if its argument is a number, either integer or real. Otherwise, the predicate *number?* returns *nil*. Example: *(number? 3.14159265359) → true*.
- The function *(list?)* is the last of the predicates seen in this chapter. It returns *true* when the supplied argument is a list and returns *nil* if its argument is not a list. Example: *(list? '(a b c)) →* true, but *(list? 'a) → nil*.
- The function *(stats)* is a function contained in the mathematical library of NewLisp. It takes a list of numbers as its argument and returns another list whose elements are statistical values corresponding to the statistical operations performed on the numbers of the list used as argument such us mean value, standard deviation, variance, etc. Example: *(stats '(-3 -2 1 4 8 11 13 12 9 6 2 -1)) → (12 5 4.833333333 5.640760748 31.81818182 0 -1.661427211)*. This function is an extended feature of NewLisp and may not be available in other Lisp dialects.
- The function *(first)* takes a list as its argument and returns its first element. Example: *(first '((a b c) x y z)) → (a b c).* This function is named *(car)* in the Common Lisp dialect.
- The function *(rest)* takes a list as its argument and returns a list composed by all the elements contained in the original list with the exception of its first one. Example: *(rest '(a b c d)) → (b c d).* This function is named *(cdr)* in the Common Lisp dialect.
- The function *(n*th*)* takes an index *i* and a list of *n* elements as arguments. It returns the element at position *i* in the list. The first element in the list is indexed as zero by (*n*th). In other words, *(nth 0 list)* returns the first element of *list*, while *(n*th *(- n 1) list)* returns the last element of *list*. Example: *(n*th *1 '(this (seems tricky) at first)) → (seems tricky).*
- The function *(last)* takes a list as its argument and returns the last element of the list supplied as argument. Example: *(last '(seems tricky)) → tricky.*

- The function *(length)* takes a list as its argument and returns the number of elements in it. Example: *(length '(this (seems tricky) at first)) → 4*, but note that *(length '(this seems tricky at first)) → 5.*
- The function *(cons)* is used for building lists. It takes two arguments, either atoms or lists, and then constructs a new list with them. Example: *(cons 'not 'hard) → (not hard)* but take care with the use of reiterated *conses* because it quickly builds nested lists: *(cons '(this is) '(not hard)) → ((this is) not hard).*
- The function *(list)* is another function for building lists. It takes any number of atoms or lists and then joins it all, returning another list. As an example, *(setq a-list (list 'a 'b 'c 'd))* and then *(list '1 '2 '3 a-list '(x y z)) → (1 2 3 (a b c d) (x y z)).*
- The function *(assoc)* takes the first argument passed to the function and then uses it as a key for searching the list passed as the second argument. If a match is found, then it returns the member list. If no match is found, it returns nil. Example: *(setq mountains '((Everest 8848) (Kilimanjaro 5895) (Mont-Blanc 4695) (Aconcagua 6962))* and then: *(assoc 'Kilimanjaro mountains)) → (Kilimanjaro 5895).*
- The function *(append)* takes a list as its first argument and then appends the second argument, which becomes the last element of the list given by the first argument. As an example: *(setq a-list '())* then *(setq a-list (append a-list '(first-element)))* and then *(setq a-list (append a-list '(second-element third-element))) → (first-element second-element third-element).* Note that *append* is not destructive, hence the continuous use of *setq* for *a-list*
- The function *(reverse)* simply reverses all the elements inside a list. Example *(reverse '(a b c) → (c b a).* Remember that this function is destructive.
- The function *(push)* adds elements into a list "pushing" them sequentially from the frontal part of the list to its back. It is a destructive function. Example: *(setq names '())* then *(push 'John names)* and then *(push 'Anna names) → (Anna John).*
- The function *(pop)* extracts and then returns the first element in the list supplied as its argument. In the previous example, *names → (Anna John)* and then (pop names) → *Anna.* Needless to say, it is also a destructive function.
- The function (now) returns data from the computer's internal clock when it is called. Example: *(now) → (2014 2 26 13 34 40 592460 56 3 -60 0).* This function is an extended feature of NewLisp and may not be available in other Lisp dialects.
- The function *(rotate)* takes a list as its first parameter and then moves every element contained in it *n* positions towards its right if the parameter *n* is positive or towards left if *n* is negative. Example: *(setq a-list '(a b c d))* then *(rotate a-list 3) → (b c d a)* and *(rotate a-list -3) → (a b c d).* This function is destructive.
- The function *(random)* returns a real number between 0 and 1. Example: *(random) → 0.7830992238*. This function is an extended feature of NewLisp and may not be available in other Lisp dialects.
- The function *(extend)* takes several lists as arguments and returns a single list formed by all the elements from the supplied lists. Example: *(setq whole-list '())* then *(setq list-1 '(a b c))* and *(setq list-2 '(d e f))*, then typing *(extend whole-list list-1 list-2) → (a b c d e f).* Since this function modifies the original list, it is a destructive function.

- Finally, the function *(flat)* analyzes the list supplied as its unique argument and then eliminates any sub-list included in it, returning a list where all the elements are atoms. Example: *(flat '(a (b (c d) (e f) g) h))* → *(a b c d e f g h)*.

A more comprehensive description of these functions and their respective parameters can be obtained consulting NewLisp's on-line help from the Menu: Help → Newlisp Manual and Reference. In this section of the chapter we have only described the simplest use of some of the functions.

References

Banks, J., et al.: Discrete-Event Simulation, Prentice Hall, Upper Saddle River (2009)
Climatedata.eu.: Climate Berlin, Germany (2014). http://climatedata.eu/climate.php?loc=gmxx0007&lang=en. Accessed Feb 2014
Sturgul, J.: Mine Design: Examples Using Simulation. Society for Mining Metallurgy and Exploration (1999)
Zhang, R., et al.: Fuzzy Control of Queuing Systems. Springer, Berlin (2005)

http://www.springer.com/978-3-319-23185-3

A Practical Introduction to Fuzzy Logic using LISP

Argüelles Méndez, L.

2016, XV, 370 p. 109 illus., 1 illus. in color., Hardcover
ISBN: 978-3-319-23185-3