Notice how the first list corresponds to the x input in the *lambda*-function, the second to y and the third to z).

Problems could be foreseen when the input-lists to *mapcar* are not of equal length. Should this occur, then *mapcar* will stop evaluating when the shortest input-list is depleted. It will do so, however, without posting an error and by returning the output that was calculated up until then. Therefore, the output of a *mapcar* with multiple input-lists will always have the same length as the shortest input-list.

17.1.3. File I/O in Common Lisp

When programming complex data structures in CL, the use of data input and output-files can become a necessity. For example, the (x,y)-coordinate lists that describe different airfoils are typically stored in simple text-files, which must be accessed and stored into memory.
Alternatively, when a certain optimization process has reached the optimum shape of some aerodynamic surface, this output should be stored on disk for subsequent post-processing.

The structure of CL allows for a number of different ways of reading an input file. A convenient way of doing so is by defining the reader as a separate function. In this fashion the specifics of how the input file should be read are defined only once, while this function can simply be called by its name and input file path. In this section a number of examples of such reader functions is given.

A simple example of such a file reader is given by the following code:

```
(in-package :gdl-user)

(defun read-points (file-name)
  (with-open-file (in file-name)
    (let (result)
      (do ((line (read-line in nil nil) (read-line in nil nil)))
          ((or (null line) (string-equal line "")) (nreverse result))
        (let ((xyz-list (read-from-string (string-append "(" line ")"))))
          (push xyz-list result))  ))))
```

Now, consider the following data file *input.dat*, (hypothetically) located in *C:\documents and settings\user\desktop\*, which has the following content, when opened by a text-editor:

```
1               0               0
0.9981          0.043577871     0
0.992403877     0.086824089     0
0.982962913     0.129409523     0
```

```
0.96984631        0.171010072      0
0.953153894       0.211309131      0
0.933012702       0.25             0
0.909576022       0.286788218      0
0.883022222       0.321393805      0
0.853553391       0.353553391      0
0.821393805       0.383022222      0
0.786788218       0.409576022      0
0.75              0.433012702      0
```

Which is a listing of 3D-points, whereby each line represents a different point and the columns represent x, y and z-values.
When the code above is compiled and thereby the function *read-points* is defined, it can be used to access the data stored in *input.dat* from the command-line:

```
(read-points “C:\\documents and settings\\user\\desktop\\input.dat”)
```

output:

```
((1 0 0)  (0.9981 0.043577871 0) (0.992403877 0.086824089 0) (0.982962913
0.129409523 0) (0.96984631 0.171010072 0) (0.9531538940 0.211309131 0)
(0.933012702  0.25 0) (0.909576022 0.286788218 0) (0.883022222 0.321393805 0)
(0.853553391  0.353553391 0) (0.821393805 0.383022222 0) (0.786788218
0.409576022 0) (0.75 0.433012702 0))
```

*(Pay attention to the way in which file-pathnames are specified in CL. Double-quotes should be used around the pathname and double back-slashes to separate the different folders in the pathname. There are other formats that also work, but this one is the most fool-proof in CL)

The output of the *read-points* function, when applied on *input.dat*, is a list of lists. The content of each line is placed in a separate list and these lists corresponding to each input line are bunched together in a new list.

By slightly altering the code, a different in-memory storage of the same input file can be obtained:

```
(in-package :gdl-user)

(defun read-points (file-name)
  (with-open-file (in file-name)
    (let (result)
      (do ((line (read-line in nil nil) (read-line in nil nil)))
            ((or (null line) (string-equal line "")) (nreverse result))
              (push line  result)
          ))))
```

When this function is compiled and used to access *input.dat*, the following result is returned:

```
(( “1     0          0”  “0.9981       0.043577871   0” “0.992403877        0.086824089   0”
“0.982962913  0.129409523   0” “0.96984631         0.171010072    0” “0.9531538940
0.211309131 0” “0.933012702          0.25     0” “0.909576022       0.286788218    0”
“0.883022222  0.321393805   0” “0.853553391       0.353553391    0” “0.821393805
0.383022222   0” “0.786788218         0.409576022   0” “0.75          0.433012702   0”)
```

This time each line of *input.dat* is stored as a string (“…”). These strings are in turn bunched together in a single list.

Next, a code is shown that will store the data of *input.dat* into a list of geometry points, which is a data-construct different from regular lists in GDL:

```
(in-package :gdl-user)

(defun read-points (file-name)
  (with-open-file (in file-name)
    (let (result)
      (do ((line (read-line in nil nil) (read-line in nil nil)))
          ((or (null line) (string-equal line "")) (nreverse result))
        (let ((xyz-list (read-from-string (string-append "(" line ")"))))
          (when (and (= (length xyz-list) 3)
                     (every #'numberp xyz-list))
            (push (surf::apply-make-point xyz-list) result)))))))
```

output:

```
(#(1 0 0)  #(0.9981 0.043577871 0) #(0.992403877 0.086824089 0) #(0.982962913
0.129409523 0) #(0.96984631 0.171010072 0)  #(0.9531538940 0.211309131 0)
#(0.933012702 0.25 0) #(0.909576022 0.286788218 0) #(0.883022222 0.321393805 0)
#(0.853553391 0.353553391 0)  #(0.821393805 0.383022222 0) #(0.786788218
0.409576022 0) #(0.75 0.433012702 0))
```

Notice how the lists containing the coordinate data are preceded by a #, which indicates that they are not ordinary lists. This was caused by the use of the surf-package’s apply-make-point function in the definition of read-points.
This function can be easily adjusted to read-in only the first two ordinates of each line, while discarding the third. This is done by replacing the third line from below in the above code as follows:

```
(when (and (= (length xyz-list) 3)      ->        (when (and (= (length xyz-list) 2)
```

After compiling this new function-definition, the output will be:

```
(#(1 0)  #(0.9981 0.043577871) #(0.992403877 0.086824089) #(0.982962913
0.129409523) #(0.96984631 0.171010072)  #(0.9531538940 0.211309131)
#(0.933012702 0.25) #(0.909576022 0.286788218) #(0.883022222 0.321393805)
#(0.853553391 0.353553391)  #(0.821393805 0.383022222) #(0.786788218
0.409576022) #(0.75 0.433012702))
```

It would go into too much detail to elaborate on the exact purpose of each line of code in these examples or to explore all possible variations. Rather, these examples are given as a starting point for the reader to experiment with.

The general lay-out of these examples is based on the *with-open-file* function. This function opens a file of which the pathname is specified in its argument. *with-open-file* doesn't perform any action on the opened file by itself, rather it keeps the file open in order for some other operations to be performed on the file. These other operations are specified after the file pathname input-argument. In all of the above examples the output (*result*) is defined (*let*) as the outcome of an iteration (*do*) that takes the lines of the input file as a stream (*in*).