

Another primitive kind of thing Lisp knows about are text strings.

```
> "Hello there!"  
"Hello there!"
```

Strings start and end with a double quote and contain all the text inbetween. Strings too, stand for themselves³.

Making lists

To represent anything more complicated, Lisp uses one very simple yet very flexible concept: the list. A list is an ordered collection of things, written as a sequence of elements, separated by spaces and surrounded by an opening and closing parenthesis.

For example,

```
(1 2 3)
```

is a list containing three elements, the numbers 1, 2 and 3. Another example,

```
("abc" "def" "ghi")
```

is again a list of three elements, this time containing the strings "abc", "def" and "ghi".

Part of the power of lists comes from the fact that the elements can be anything, including other lists.

For example,

```
((1 2 3) ("abc" "def" "ghi"))
```

is a list of two elements: the first of these elements is another list, containing the numbers 1, 2 and 3, the second of these elements is also another list, containing the strings "abc", "def" and "ghi".

Lisp uses lists for two purposes: to build data structures and to represent programs. This unique aspect gives Lisp great power and flexibility. Say you want to write down, represent, record or store the following data:

- `firstname = John`
- `lastname = McCarthy`
- `birthday = 19270904`

The shortest solution would be to agree to always keep the data in this order and simply list it:

```
("John" "McCarthy" "19270904")
```

Although short and efficient, it might be error-prone and hard to extend. In Lisp we often add a descriptive tag to each data element:

```
((firstname "John")  
 (lastname "McCarthy")  
 (birthday "19270904"))
```

or

```
(firstname "John"  
 lastname "McCarthy"  
 birthday "19270904")
```

All three solutions are frequently used⁴. Notice how we split the lists across a number of lines and how we indent successive lines so that we get vertical alignment of corresponding elements: this is important to improve readability.

It's more fun to compute

To add two numbers, like 1 and 2 together, as in $1 + 2$, we write the following Lisp code:

```
> (+ 1 2)
3
```

When Lisp sees a list, it will generally look at its first element to determine what to do. In this case, it sees the plus sign which it knows it refers to the procedure for addition. The other two elements in the list are seen as the arguments, the numbers to be added together. Lisp will look at each argument to find out what it means. Numbers stand for themselves.

Actually, many Lisp functions and procedures accept any number of arguments, if that makes sense:

```
> (+ 10 20 30 40 50)
150
```

Common Lisp in its standard definition contains well over a thousand different functions and procedures. Here are some examples:

```
> (* 7 7)
49
> (expt 7 2)
49
> (sqrt 49)
7
```

A special class of functions are predicates: these test whether a certain property is true or not. Conventionally, most of them have the letter *p* appended to their name:

```
> (oddp 7)
T
> (evenp 7)
NIL
> (= 7 7)
T
```

The special value *T* stands for 'true' in Lisp. The special value *NIL* stands for 'false' in Lisp.

There is much more to Lisp than just mathematics though. Assuming we have a file named 'test.txt' in the '/tmp' directory on our computer, we can do some file manipulation as follows:

```
> (probe-file "/tmp/test.txt")
#P"/tmp/test.txt"
> (delete-file "/tmp/test.txt")
#P"/tmp/test.txt"
> (delete-file "/tmp/test.txt")
NIL
```

The function *PROBE-FILE* tests whether a certain file exists. If it exists it returns how it would internally represent the file's pathname. In Common Lisp, everything that is not *NIL* is considered to be 'true', that is why *PROBE-FILE* is also a predicate: by returning something that might also be useful in other cases it indicates that it actually found the file to exist.

The function *DELETE-FILE* will delete a file, if it exists. If the deletion was successful, i.e. the file existed and you are allowed to delete it, Lisp will return 'true' by returning the internal file's pathname. Deleting a file that doesn't exist simply returns *NIL* for 'false'.

Infinite combinations

It is also possible to combine several computations, by simply nesting them. If we want to compute $(5 + 5) * (10 + 10)$ we write:

```
> (* (+ 5 5) (+ 10 10))
200
```

On the other hand, to compute $2 * (4 + 6)$ we write:

```
> (* 2 (+ 4 6))
20
```

When Lisp sees either of the previous two examples it knows that it has to do a multiplication, by looking at the first element. Next it has to deal with the arguments. This is done by computing the meaning each one. This is necessary because we need numbers to multiply. If an argument is again a list, the process starts all over.

We can write out explicitly what goes on inside Lisp for each computation above:

```
> (* (+ 5 5) (+ 10 10))
> (+ 5 5)
> 5
5
> 5
5
10
> (+ 10 10)
> 10
10
> 10
10
20
200
```

```
> (* 2 (+ 4 6))
> 2
2
> (+ 4 6)
10
20
```

As you can see, in order to do the multiplication, the addition(s) must be computed. In order to do the addition(s), the meaning of the numbers has to be determined. Numbers stand for themselves.

We already saw that some functions accept any number of arguments, like:

```
> (< 1 2 3 4)
T
> (- 100 5 5 5 5)
80
> (max 34 7 88 12)
88
> (min 34 7 88 12)
7
> (- 10)
-10
> (+ 10)
10
> (+)
0
```

```
> (* 10)
10

> (*)
1
```

The examples with one argument might seem odd or useless, but are easy to understand. You probably remember from your mathematics classes what the 'identity element' is for addition and multiplication: adding zero to something makes no difference, like multiplying something with one makes no difference. This explains the last two examples with no arguments.

Some Lisp function accept optional arguments of a different kind: they allow you to do more with the same function. These arguments are tagged by their name, which makes the code easier to read, the arguments easier to remember, their positions interchangeable, and each of them is optional against the others. Most of these arguments have default values when they are absent. For example:

```
> (parse-integer "123")
123

> (parse-integer "FF" :radix 16)
255

> (parse-integer "12-345-67" :start 3 :end 6)
345
```

The function PARSE-INTEGGER takes a string as argument and tries to convert (parse) it to an integer. By default it assumes that you will give it a decimal (base 10 or radix 10) number. The `:radix` option (which defaults to 10) can be used to tell it to expect an hexadecimal number by specifying 16. In some cases, you might want to parse only a part of your string. Specifying the `:start` and `:end` options allows you to do just that. Note that Common Lisp uses zero based indexing.

Remember me

We now know about numbers, strings, T and NIL. We know about invoking and composing functions. Like any programming language Lisp also supports the concept of variables. These are named places that can contain a value. There exist some predefined variables, for example:

```
> pi
3.141592653589793
```

We can make our own variables just by using them, by giving them a value:

```
> (setf a 100)
100

> (setf b 200)
200

> a
100

> b
200

> (+ a b)
300
```

Think of SETF as being a powerful assignment operator, read it as a = 100 and b = 200. You can see that SETF always returns the value it last assigned. You refer and use the value of a variable simply by using the variable's name. When Lisp sees a variable's name, it gets the variable's value and uses that as its meaning.

```
> c
Error: the variable c has no value.

> (setf c (+ a b))
300

> c
300

> (setf c 0)
0

> c
0
```

It is an error if you refer to a variable that hasn't been given a value. Variables keep their values for the duration of the Lisp session. You can give existing variables new values.

And I quote

Remember how we talked about lists being used both for program code and data structures? Suppose we want to assign the list (1 2 3) to the variable x:

```
> (setf x (1 2 3))
Error: 1 is not a function name.
```

What happened? Lisp looked at our list (1 2 3) and tried to use the first element, 1, as a function to apply to the other elements, 2 and 3. This is not what we meant. What we meant is that Lisp should see the list (1 2 3) as a list, not a function call. This is a common situation in Lisp and there is a special notation for it: precede it with a quote.

```
> (setf x '(1 2 3))
(1 2 3)

> (length x)
3

> (reverse x)
(3 2 1)

> x
(1 2 3)
```

The quote means to Lisp: stop thinking and take what follows literally. Lisp contains a large number of functions to work with lists: LENGTH and REVERSE are but two examples. Other functions take lists apart:

```
> (first '(a b c))
a

> (second '(a b c))
b
```

```
> (elt '(a b c) 1)
b

> (rest '(a b c))
(b c)
```

Given the list (a b c) as argument, the functions FIRST and SECOND do the obvious thing. The function ELT returns an element at a certain index, the first one being at index 0. The counterpart of FIRST is REST: it returns everything but the first element.

You might be wondering what type the elements are of the list (a b c) as used above. They are not numbers, nor are they strings. They are symbols. Think of symbols as pure, unique names. Symbols are used in data structures, but also in code to denote functions, variables and other elements. To tell Lisp you mean a symbol you have to quote it, otherwise Lisp will think it is a variable. Some examples:

```
> 'a
a

> (setf x 'a)
a

> (eql x 'a)
T

> (find 'b '(a b c))
b

> (find 'z '(a b c))
NIL

> (position 'b '(a b c))
1
```

```
> (count 'c '(a b b c c c))
3
```

Note how each time we want to talk about a symbol we have to quote it. The function FIND looks up an item in a list and returns it if found. The function POSITION does almost the same thing, but returns the index at which it finds the item. Finally, the function COUNT counts the number of occurrences of an item in a list.

Remember how we used list data structures to store some information earlier? Here are two examples showing how easy it is to access list based information:

```
> (assoc 'lastname
        '((firstname "John")
          (lastname "McCarthy")
          (birthday "19270904")))
(lastname "McCarthy")

> (getf '(firstname "John"
                lastname "McCarthy"
                birthday "19270904")
        'firstname)
"John"
```

For historical reasons ASSOC and GETF are not symmetrical in how you pass them arguments or in what they return, but they both do their job: find the tagged data element that you are looking for.

Lists can also be used to represent mathematical sets (unordered collections of things where each thing occurs only once). Consider the following interactions with Lisp:

```
> (setf set1 '(a b c))
(a b c)

> (setf set2 '(c d e))
(c d e)

> (intersection set1 set2)
(c)

> (union set1 set2)
(e d a b c)

> (set-difference set1 set2)
(b a)

> (set-difference set2 set1)
(e d)
```

Again, the functions INTERSECTION, UNION and SET-DIFFERENCE should speak for themselves, if you remember your elementary mathematics from long ago.

One important loose end we did not yet talk about is the empty list. In Lisp there are two ways to represent the empty list: as a list with no elements, `()` or as symbol, namely `NIL`. Quoting either is allowed but makes no difference as they stand for themselves.

```
> '()
NIL

> (length '())
0

> (length NIL)
0

> (find 'x '())
NIL

> (union '(1 2 3) '())
(1 2 3)
```

The length of the empty list is of course zero. Looking for something in an empty list always returns false or `NIL`. For the set operation union, the empty list is the identity element.

Powerful spells

Two features that make Lisp powerful are its ability to operate on whole lists of items at once and to work with functions as if they were data. To be able to present some concrete examples, we'll be using some functions from an add-on library. See appendix 2 for more information on this. To keep things simple, we'll act as if these extra functions came built in to Lisp.

```
> (list-directory "/img/nature/")
(#P"/img/nature/aurora.jpg"
 #P"/img/nature/clown-fish.jpg"
 #P"/img/nature/dew-drop.jpg"
 #P"/img/nature/earth.jpg"
 #P"/img/nature/flowing-rock.jpg"
 #P"/img/nature/ladybug.jpg"
 #P"/img/nature/rock-garden.jpg"
 #P"/img/nature/rocks.jpg"
 #P"/img/nature/snowy-hills.jpg"
 #P"/img/nature/stones.jpg"
 #P"/img/nature/water.jpg"
 #P"/img/nature/zebra.jpg")

> (mapcar 'file-size
         (list-directory "/img/nature/"))
(1378680 654257 2712436 1860714 932143 2529205
 3924771 2221066 2386322 1294792 2114546 3940790)

> (mapcar 'pathname-name-and-type
         (list-directory "/img/nature/"))
("aurora.jpg" "clown-fish.jpg" "dew-drop.jpg"
 "earth.jpg" "flowing-rock.jpg" "ladybug.jpg"
 "rock-garden.jpg" "rocks.jpg" "snowy-hills.jpg"
 "stones.jpg" "water.jpg" "zebra.jpg")

> (sort * 'string>)
("zebra.jpg" "water.jpg" "stones.jpg"
 "snowy-hills.jpg" "rocks.jpg" "rock-garden.jpg"
 "ladybug.jpg" "flowing-rock.jpg" "earth.jpg"
 "dew-drop.jpg" "clown-fish.jpg" "aurora.jpg")
```

It is pretty clear what LIST-DIRECTORY⁵ does: given a directory name, it returns a list of the files in it. We want to get the file size of each of those files. The function MAPCAR takes two arguments, the name of a function and a list. It will then apply the function on each element of the list and return a list of the results. The second example uses FILE-SIZE⁶ to do the actual work. Similarly, PATHNAME-NAME-AND-TYPE⁷ returns just the name and type without the parent directory stuff. The result is a list with the simple filenames.

Finally, we use the SORT function to sort this list inverse lexicographically. Note the * argument. This is a very handy special variable that is always bound to the result of the last computation. In our case, the list of simple filenames. In a directory with both files and subdirectories, we might want to filter each type.

```
> (remove-if 'directory-pathname-p
            (list-directory "/img/"))
(#P"/img/aqua-blue.jpg"
 #P"/img/aqua-graphite.jpg"
 #P"/img/flow-1.jpg"
 #P"/img/flow-2.jpg"
 #P"/img/flow-3.jpg"
 #P"/img/lines-blue.jpg"
 #P"/img/lines-graphite.jpg"
 #P"/img/lines-moss.jpg"
 #P"/img/lines-plum.jpg"
 #P"/img/ripples-blue.jpg"
 #P"/img/ripples-moss.jpg"
 #P"/img/ripples-purple.jpg"
 #P"/img/tiles-blue.jpg"
 #P"/img/tiles-pine.jpg"
 #P"/img/tiles-warm-grey.jpg")
```

```
> (remove-if-not 'directory-pathname-p
                (list-directory "/img/"))
(#P"/img/abstract/"
 #P"/img/black-white/"
 #P"/img/nature/"
 #P"/img/plants/"
 #P"/img/solid-colors/")
```

The function REMOVE-IF takes a function and a list as argument, and returns a list containing only those elements for which the function yields true. The complementary REMOVE-IF-NOT selects those elements for which the function yields false. The predicate we are using, DIRECTORY-PATHNAME-P⁸ is a test that returns true when the argument is a directory. Another Lisp technique is to use a list as data for a computation. Suppose we want to find the largest and smallest file size in our directory.

```
> (setf file-sizes
      (mapcar 'file-size
              (list-directory "/img/nature/")))
(1378680 654257 2712436 1860714 932143 2529205
 3924771 2221066 2386322 1294792 2114546 3940790)

> (apply 'max file-sizes)
3940790

> (apply 'min file-sizes)
654257

> (apply '+ file-sizes)
25949722
```

Remember how MIN and MAX took any number of arguments ? We are using that feature now. The function APPLY takes two arguments: a function and a list. It will call the function using the list as the list of argument to the function. To save some typing, we temporarily saved our list of file sizes in a variable named file-sizes. The last example adds all file sizes together⁹.

Define that, please

Like any programming language, Lisp allows you to define new functions. We want a function that tests whether a file is a JPG file:

```
> (defun pathname-jpg-p (pathname)
    (string-equal (pathname-type pathname) "jpg"))
pathname-jpg-p

> (pathname-jpg-p "/img/flow-1.jpg")
T

> (pathname-jpg-p "/img/nature")
NIL
```

There are three parts to a function definition:

- the name
- the parameter list
- the code

A Common Lisp function definition is a list that starts with DEFUN. Next is the name of the new function, PATHNAME-JPG-P in this case. The third element is the parameter list, there is only one parameter, pathname, here. The rest of the list is the definition code proper (also called the body).

When a function is called, Lisp will look up its definition. The argument list is matched to the parameter list. Then the body is executed with each parameter assigned to its corresponding argument. Finally the function ends and the result is returned.

In our example, pathname gets assigned to the filename string we pass as argument. Then the body code is executed: (string-equal (pathname-type pathname) "jpg"). The function PATHNAME-TYPE extracts the extension from a filename. With STRING-EQUAL we test whether the extension is equal to "jpg" or "JPG". The result of this last test is at the same time the result of the function.

We have, of course, only scratched the surface of computer programming here, let alone the many different programming techniques possible in Common Lisp. Going further is well beyond the scope of this introduction. Let's end this part about function definitions with another example. We give two different implementations of a function called LIST-LARGE-FILES that given a directory and a minimum size returns only files that are larger:

```
> (defun list-large-files (dir min-size)
  (let ((large-files '()))
    (dolist (pathname (list-directory dir))
      (if (<= min-size (file-size pathname))
          (push pathname large-files)))
    larg-files))
list-large-files

> (defun list-large-files (dir min-size)
  (loop :for pathname :in (list-directory dir)
        :when (<= min-size (file-size pathname))
        :collect pathname))
list-large-files

> (list-large-files "/img/nature/" 2000000)
(#P"/img/nature/dew-drop.jpg"
 #P"/img/nature/ladybug.jpg"
 #P"/img/nature/rock-garden.jpg"
 #P"/img/nature/rocks.jpg"
 #P"/img/nature/snowy-hills.jpg"
 #P"/img/nature/water.jpg"
 #P"/img/nature/zebra.jpg")
```

The second implementation uses the advanced LOOP construct which makes it easy to read. A disadvantage is that LOOP forms effectively a language within a language with its own rules. The first implementation uses more classical Lisp constructs, like LET to introduce temporary local variables, DOLIST to iterate over list elements and PUSH to add elements to a list. Note how indentation and syntax coloring are used to make larger chunks of Lisp code readable.

Powerful spells revisited

Remember how we used powerful mapping functions that operated on whole lists at once by applying a function to each element? In those examples, we used already existing functions like FILE-SIZE or DIRECTORY-PATHNAME-P. In the previous section we learned how to define our own functions, like PATHNAME-JPG-P. Once we define our own function we can use it:

```
> (remove-if-not 'pathname-jpg-p
  (list-directory "/img/"))
(#P"/img/aqua-blue.jpg"
 #P"/img/aqua-graphite.jpg"
 #P"/img/flow-1.jpg"
 #P"/img/flow-2.jpg"
 #P"/img/flow-3.jpg"
 #P"/img/lines-blue.jpg"
 #P"/img/lines-graphite.jpg"
 #P"/img/lines-moss.jpg"
 #P"/img/lines-plum.jpg"
 #P"/img/ripples-blue.jpg"
 #P"/img/ripples-moss.jpg"
 #P"/img/ripples-purple.jpg"
 #P"/img/tiles-blue.jpg"
 #P"/img/tiles-pine.jpg"
 #P"/img/tiles-warm-grey.jpg")
```

It is a bit odd though to define a new function just to do some filtering, especially when this function is maybe used only once. The solution is to write an anonymous function definition where we put the name before:

```
> (remove-if-not (lambda (p)
                 (string-equal (pathname-type p)
                               "jpg"))
                (list-directory "/img/"))
```

An anonymous function is a list that starts with LAMBDA (from the greek letter), the rest is like a DEFUN function definition, with a parameter list and a body of code. The lambda construct is much more powerful than can be seen from this example however. To continue from our example, we removed the explicit separate definition and moved to an anonymous inline definition. That is great, but suppose we want to test for JPG here, TXT there and GIF somewhere else. Would it not be nice if we could somehow get some help and save some typing in constructing these test functions. Using LAMBDA we can write functions that return new on-the-fly generated functions.

```
> (defun pathname-type-test (extension)
  (lambda (pathname)
    (string-equal (pathname-type pathname)
                  extension)))
pathname-type-test

> (pathname-type-test "jpg")
#<anonymous function 21D48B0A>

> (funcall (pathname-type-test "jpg")
           "/img/aqua-blue.jpg")
T

> (funcall (pathname-type-test "txt")
           "/img/aqua-blue.jpg")
NIL

> (remove-if-not (pathname-type-test "jpg")
                 (list-directory "/img/"))
```

Our helper function PATHNAME-TYPE-TEST is a function that takes one argument, an extension string. It returns a new anonymous function of one argument, pathname. This function, when invoked on

a pathname will test if its type equals the extension. Different calls to PATHNAME-TYPE-TEST will result in different independent functions that each do their own test. Similar to APPLY, FUNCALL takes a function and calls it on its argument(s). When Lisp expects a function, it will take either a name of a defined function or an anonymous lambda. The anonymous lambda can be written inline or come from a function call, as in PATHNAME-TYPE-TEST. Lambdas can also be stored in variables and data structures.

Where to go from here ?

This concludes our introduction to Common Lisp. We have now seen the most important and most basic Lisp constructs. We learned:

- how Lisp reads our input text, computes and returns a result
- about primitive, literal data like numbers, strings, symbols, T and NIL.
- about lists and how they are used for both data structures and for code.
- how lists can be nested to build more complex structures
- about functions and function calls
- how Lisp computes nested function calls
- about variables, setting and using their values
- how quoting forces Lisp to see certain lists and symbols as data
- how functions can be used to operate on whole lists at once
- how to define new functions
- how to use lambda to work with anonymous functions

If this introduction convinced you of the elegance, flexibility and power of Lisp, then you are ready to move on and to read a more advanced introductory text. Consult the bibliography section for some good titles. Appendix 1 tries to get you started on getting a Common Lisp system to install on your computer so that you experiment a bit.

Appendix 1: Getting Common Lisp

Common Lisp is, unlike many other popular computer programming languages, properly standardised by ANSI X3.226-1994. This means that there are many compatible implementations. There is no dominant, de facto or reference implementation - compare the situation to C or C++.

A very good commercial implementation is LispWorks. Its graphical IDE, available on Windows, Linux and Mac OS X, makes it very attractive for beginning as well as seasoned Lisp programmers. A special version called LispWorks Personal Edition can be downloaded free of charge.

<http://www.lispworks.com>

Another well known, multi-platform commercial implementation is Franz's Allegro Common Lisp, which has an IDE on some platforms. A free trial version of ACL is available as well.

<http://www.franz.com>

There are many high quality, open source Common Lisp implementations available. Here is a list of web addresses for CLISP, SBCL, CMUCL, OpenMCL and ECL respectively:

<http://clisp.cons.org/>
<http://www.sbcl.org/>
<http://www.cons.org/cmucl/>
<http://openmcl.closure.com/>
<http://ecls.sourceforge.net/>

Two 'getting started' initiatives are worth mentioning: the Lisp In A Box distribution and the LispWorks STARTER-PACK. Both aim to help beginners set up a working Lisp environment.

<http://common-lisp.net/project/lispbox/>
<http://weitz.de/starter-pack/>

Appendix 2: The cl-1st-contact library

To support the examples in Common Lisp: First Contact, a small library was created. By loading this library you can run the examples yourself. Like most modern CL libraries, cl-1st-contact uses the ASDF mechanism compilation, loading and possibly installation.

The main web page for this document and for the library is

<http://homepage.mac.com/svc/CommonLispFirstContact>

You can get started with ASDF by reading

<http://constantly.at/lisp/asdf/>
<http://common-lisp.net/project/asdf-install/tutorial/>
<http://common-lisp.net/~mmommer/asdf-howto.shtml>
<http://www.cliki.net/asdf>

If you are not familiar with ASDF and/or don't have it configured properly or if you just want to play with the code without installing anything, just load the file LOAD-ME.LISP and you are done.

Bibliography

There are many books about Common Lisp. Next is a selection of some good books to learn Common Lisp. Most of these books are available online.

Practical Common Lisp

Peter Seibel, Apress, 2005

If you buy one book, it should be this one. This modern introduction to Common Lisp is ideal for people familiar with other programming languages. The author comes from a Java background himself and it shows in his approach and technical explanations. The examples are practical, modern, down to earth and very useful. This book is definitively interesting for seasoned Lisp programmers as well.

Common Lisp: A Gentle Introduction to Common Lisp

David S. Touretzky, Benjamin-Cummings Pub Co, 1989

A typical college style Lisp course style book. Well written and very clear with a strong emphasis on simplicity. Covers a large part of the Common Lisp language. Especially useful for beginners.

ANSI Common Lisp

Paul Graham, Prentice Hall, 1995

If you want a quick, no frills introduction to Lisp, this is the book. Contains a very useful quick reference section to everything inside Common Lisp.

Common Lisp: An Interactive Approach

Stuart C. Shapiro, W.H. Freeman & Company, 1991

Another college style introduction with lots of examples and exercises. Contains a sizeable reference section as well.

Basic Lisp Techniques

David J. Cooper, Jr., Franz Inc, 2003

A predecessor to Practical Common Lisp: a hands on, modern and very practical introduction to Lisp.

Successful Lisp: How to Understand and Use Common Lisp

David B. Lamkins, BookFix, 1995,2004

Recently republished, this book constitutes yet another approach to a Lisp introduction. Contains both a faster, lesson based, intro track and then later expands into many in depth chapters.

Acknowledgements

I thank Nicky Peeters for proof reading this text and for his feedback. Many thanks to Jens Teich, Andrew Philpot and Baishampayan Ghose for their feedback and for helping to improve this text.

Colofon

This document was written on a Apple MacBook Pro running Mac OS X using the Pages word processor. The main text and titles font is Helvetica. The code font is Andale Mono. The front title font is Futura. The examples were run using LispWorks and CLISP.

For feedback please email to `svc <at> mac.com`

Endnotes

¹ There are many excellent introductions and books for learning Common Lisp. They all become quite technical and complex early on: they kind of have to, because they want to teach you how powerful and flexible Lisp is. Teaching how Lisp really works also requires long and deep explanations. This introduction tries to confront the reader gently with Lisp, explaining the necessary concepts in a cautiously selected order. I try to never lie, but certainly don't always tell the whole truth because that would make things too complex. I try to avoid technical and Lisp specific jargon as much as possible.

² In the course of its actions or computations, Lisp might print additional output as well.

³ To write down a string which contains a double quote, you have to escape it using a backslash: "This is a double quote: \", you see?".

⁴ Common Lisp supports all modern data structures, like arrays, hashtables as well as a complete object system.

⁵ LIST-DIRECTORY is not part of standard Common Lisp. It comes with the CL-1ST-CONTACT library, but is technically speaking part of CL-FAD, maintained by Edi Weitz. See <http://www.weitz.de/cl-fad>. There does exist a standard function called DIRECTORY that is similar.

⁶ FILE-SIZE is part of the CL-1ST-CONTACT library. It is based on the standard function FILE-LENGTH that expects an open stream rather than a pathname as argument.

⁷ PATHNAME-NAME-AND-TYPE is also part of the CL-1ST-CONTACT library. Note that it returns a string, not a pathname.

⁸ DIRECTORY-PATHNAME-P comes with the CL-1ST-CONTACT library but is again actually part of CL-FAD.

⁹ Technically, this technique might hit an implementation limit called CALL-ARGUMENTS-LIMIT: the maximum number you are allowed to pass to a function. A better solution is to use REDUCE: (reduce 'max file-sizes).