CHAPTER 2
Basics

Making a start

This chapter introduces enough basic concepts that you can start to write simple Lisp programs for yourself. I'll cover the execution model and the syntax you need both to call functions and define your own, and I'll present some common data types to get you started. Although the syntax is "unusual" much of this might not look all that earth-shatteringly "different". Think of it as part of the ground we have to cover before we can get around to the interesting bits.

So let's make a start, at the Lisp prompt (Chapter 1):

```
CL-USER>
```

What shall we say to it?

Simple Arithmetic

Lisp systems are interactive. Type some Lisp at the prompt - the system will go away, think about it and hopefully spit out some Lisp value in exchange. Then it gives you another prompt and the process repeats itself. More formally, we are in a loop (the *read-eval-print* loop or *REPL*, also widely referred to as the Lisp *listener*) which cycles round doing the following:

1. Read in a Lisp *form*.
2. *Evaluate* this form to *return* a result.
3. Print the result.

What is a *form*? It's any Lisp value you might wish to evaluate! Here are some examples:

```
42

"Hello, World"

pi
```

```
(+ 1 2 3)

(loop until (I-understand this)
      do (read-it-again))
```

Now, it happens that numbers always *evaluate* to themselves: we say that numbers are *self-evaluating*. Whenever you type in a Lisp number you will always get that same number back. By the way, Lisp offers the following types of number:

- Integers with arbitrary magnitude such as `-1461136314753` or the hexadecimal `#xdeadbeef`
- Ratios with arbitrary precision such as `1/2` or `1234567/890123343456`
- Floats such as `3.14159` or `13.73e9` typically with a choice of precisions, e.g. 64-bit IEEE format
- Complex numbers such as `#c(0 1)`, the square root of `-1`.

Exercise

λ Try typing some numbers into the prompt (press `Return` after each one).

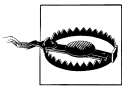
To make our calculator a little bit more interesting, we can apply arithmetic operations to these numbers. The syntax for doing so is:

1. Left parenthesis
2. Name of *operator*
3. Various *arguments* to that operator
4. Right parenthesis

For example:

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

In this case, the operator is `+` and the arguments are the numbers `1`, `2`, and `3`.



The operator is *inside* the parentheses, there are *no commas* to separate the arguments, there is *no trailing semicolon*.

If you're coming to Lisp from almost any other language you might find this arrangement perplexing to start with. We'll see later in the book (Chapter 10) that this syntax isn't an idle whim; indeed it's a powerful feature which give us access to one of Lisp's principal strengths. For now, practice!

Exercise

λ Add some numbers together.

Here are four more operators:

`-`
: takes its first argument and subtracts from that all following arguments;

`*`
: multiplies all of its arguments together;

`/`
: takes its first argument and divides that by all the following ones;

`expt`
: raises its first argument to the power of its second argument.

Note that you can mix numerical types (e.g. integer and float) within a call to any of the above and Lisp will (a) not complain and (b) do the intelligent thing on your behalf. For example:

```
(+ 1/5 0.8)
=>
1.0
```

I use "`=>`" to mean "this is what Lisp returns". So in the above, if you type in the Lisp form `(+ 1/5 0.8)` and press `Return`, I claim that a Lisp system will return and print out the number `1.0`:

```
CL-USER> (+ 1/5 0.8)
1.0
CL-USER>
```

Don't forget to press `Return` after the closing parenthesis. Otherwise Lisp won't know you've finished typing.

When Lisp signals an error (and if it hasn't already, it certainly will when you work through the next exercise), turn back to "Dealing with Errors" on page 0 for help dealing with it.

Exercise

λλλ The operators `+ - * /` all take a "variable" number of arguments. Experiment to find out what each of these does with exactly one argument. Now try each function with zero arguments, e.g. `(+)` and find out what happens.

Nested Calculations

A key point about operators is that, as with the "functions" of mathematics or other programming languages, they are constructs into which values are fed and out of which other values are returned. Common Lisp is not a "pure functional language". Its operators, whether functions or otherwise, are permitted to have side-effects and many do. But the execution model is that you do *everything* by invoking operators, passing values in, getting values back.

I've made free use of the word *operator* without telling you anything about it. Lisp operators come in three different flavors:

- *functions*
- *macros*
- *special operators*

I'll cover most of what you need to know about function calls in this chapter. I won't explain much about the other two just yet or even say anything about the difference between them until later in the book (Chapter 10).

All the operators introduced so far turn out to be functions. (More generally: every operator in this book is a function unless I explicitly tell you otherwise.) You probably wouldn't be surprised to hear that an operator called—say—`sqrt` was a function, but for `+` and friends the notion can take some getting used to. As with the parenthesis syntax, all this has a brilliant purpose and we'll get around to it later.

Exercise

λλ Use the function `sqrt` to generate a complex number. Type one back in to confirm that it's self-evaluating. Square an imaginary number.

You can nest any function call within another one (and so on, recursively, as deep as you like). For example:

```
(+ (* 2 3) (* 4 5 6))
```

There is a simple rule for this. It's universal and unambiguous:

1. Process the arguments to the function, in order "from left to right", evaluating each one in turn.

2. Once all the arguments have been evaluated, invoke the original function with these values.
3. Return the result.

Exercise

:λ Where's the recursion in the rule for evaluating function calls?

Exercise

Consider `(+ (* 2 3) (* 4 5 6))` and list all the steps required to evaluate it: what is evaluated when, the function calls and their arguments. (When written out in full, this looks obscenely long-winded. On the other hand, it's not usually you doing the work. Just bear in mind that Lisp will evaluate everything in sight, going from left to right, and evaluating inner expressions before outer ones. Think of it as the obvious thing to do.)

Not Everything is a Function

Common Lisp comes pre-equipped with 636 operators which are functions and 116 which aren't. We know that *functions always evaluate all their arguments* (in order). This needn't be true for macros and the special operators: a small number do evaluate all their arguments but it turns out that many don't. Let's meet one.

An invocation of the *special operator* `if` consists of:

1. The *test form*: a form which is always evaluated and whose value will be interpreted as either "true" or "false".
2. The *then form*: evaluated when the first value is "true".
3. The *else form*: evaluated when the first value is false.

Actually, the "else form" is optional. Leaving it out means: if the first value is false do nothing.

An example:

```
(if (saucepan-is-cool-enough)
    (pick-it-up)
  (let-it-cool-down))
```

Exercise

λλ It's just as well that `if` is not a function. If it were, you'd always get burned. Explain this.

In some ways, the invocation of a "non-function" very much looks like a function call. At this stage, consider the difference to be that the arguments aren't necessarily evaluated. The parenthesis syntax works everywhere and so we still have the pattern:

```
(operator arguments ...)
```

And, as with a function call, we can still expect to obtain return values for use later in the program (for example, we might pass them to some other operator). The rule for `if` is that it returns whatever the "then form" or "else form" returned.

Note that names in Lisp can be hyphenated and they often are. Lisp does not use infix syntax and there's no possibility of Lisp mistaking `saucepan-is-cool-enough` for arithmetic subtraction. Stay away from underscores (`saucepan_is_cool_enough`) and mixed case (`saucepanIsCoolEnough`)—while the language certainly permits such styles there's nothing to be gained in using them and you'll only draw attention to yourself.

What is Truth?

There's one value in Common Lisp which is "false". Everything else denotes logical truth. Here's the false value. It's the first value we've met which wasn't a number. Like numbers, it's "self-evaluating":

```
CL-USER> nil
NIL
CL-USER>
```

So if `(saucepan-is-cool-enough)` doesn't return `nil`, we'll `(pick-it-up)`.

Note by the way that what I typed is—by convention—lowercase and the result printed by the REPL is—by default—uppercase.

Think of Common Lisp as case-insensitive. This isn't actually true but it's good enough to get us through to Chapter 5.

I've said that anything non-`nil` is logically true. To prevent a total free for all, *predicates*—functions which are thought of as usefully returning "true" or "false"—often return the symbol `t` to denote truth. In general they're not obliged to do this—it's up to your Lisp system's implementors—so don't count on it. The important thing is whether or not the value concerned was `nil`.

Exercise

λ Find out whether `t` is self-evaluating.

A purely stylistic issue: the names of Lisp predicates generally end in the letter "p". If the name contains hyphens already, then the p is hyphenated also. So:

`zerop`
: Returns true if its argument (a number) is zero.

`array-has-fill-pointer-p`
: Returns true if its argument (an array) has a fill-pointer.

Let's take a look at some predicates, starting with = which is an immediate exception to the above naming convention (sorry about that). This function takes any number of arguments—which should all be numbers—and returns some true value provided they are all numerically the same:

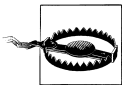
```
(= 3 3.0)        => true
(= 5/2 2.5)      => true
(= 3 3 3 3 3)    => true
(= 3 3 17 3 3)   => NIL
```

Similarly, the function < takes one or more numbers and returns *true* if they are in strictly increasing order:

```
(< 3 pi)          => true
(< 3 2.71828183)  => NIL
```

Exercise

λ Either guess what /=, >, <= and >= do or find out.



= and friends only work with numbers. We'll meet other comparison predicates later.

Exercise

λλ (Experimentally) what happens if you call = with just one argument? With no arguments? With an argument which is not a number?

Exercise

λλ How many function calls will occur in the following and what does each one return? What does the `if` return? Explain why the apparent division by zero does not take place.

```
(if (= (+ 2 2) 4)
    (* 2 2)
  (/ 1 0))
```

Exercise

λλ Look up the definition of `if` (for instance, in the HyperSpec or CLtL2 or both).

Defining Your Own Functions

Take a look at this:

```
CL-USER> (defun average-2 (this that)
           (* (+ this that)
              0.5))
AVERAGE-2
CL-USER> (average-2 15 23)
19.0
CL-USER>
```

I've defined a new function. In addition to Common Lisp's 636 built-in functions I've now got one of my own. Once it's in the system there's not a lot to tell it apart from the other 636. In particular the syntax, invocation style and evaluation model are exactly the same.

The new definition is there to stay, until either it's overwritten with another definition of the same function, or it's explicitly removed, or I quit Lisp—think of definitions as *accumulating* during the session. (I'll cover working with source files in Chapter 4 and the idea of saving a working session in Chapter 28.)

Let's figure out how function definitions work. The *signature* of the *macro* `defun` is:

```
(defun name (arg-1 arg-2 ...) form-1 form-2 ...)
```

- `Name` is the name of the function I'm defining.
- `Arg-1`, `arg-2`, ... are the *arguments* to that function.
- `Form-1`, `form-2`, ... are Lisp forms for the function to evaluate (in order).

The function returns whatever the last form returned.

Lisp doesn't need you to declare the types of a function's arguments, or of what it returns. Lisp always knows the type of any object it can reference and you're under no obligation to help it out. (There may be performance considerations here and, if you need to, you may declare types to assist the compiler. This simply isn't necessary in "most" code. Don't contemplate doing it until you've read Chapter 20.)

My function `average-2` takes two arguments (`this` and `that`). Inside the *body* of the function—that is to say, while evaluating the form `(* (+ this that) 0.5)`—the *vari-*

ables `this` and `that` take on the two values passed to `average-2`. We say that these *local variables* have been *bound* to those values. Evaluating a bound variable produces its current value.

When the function exits the *bindings* go away: `this` and `that` no longer have any value. Common Lisp bindings are by default *shallow*: the functions (`+` and `*`) called by `average-2` do not have access to the variables `this` and `that`. If you want send data to a "sub-function", pass that value as an argument!

Exercise

λλ Is the operator a function? How do you know that (other than by looking it up)?

Practice!

Exercise

λλ Write and test the function `average-3` corresponding to `average-2` but taking three arguments. Obviously, what we'd really like is a function which could average any number of values. Don't panic; Lisp has a few tricks for functions with variable numbers of arguments. We'll get to these by and by.

Exercise

λλλ Implement and test a function which takes four arguments, representing the (x, y) co-ordinates of two points, and returns the distance between them.

Exercise

λλ Find out what happens if you invoke a function with the wrong number of arguments.

Exercise

λλ Write a function which takes one argument and returns it as-is. You've only met a couple of non-number values so far but you can certainly test it out on both of these. You could also try it with

```
"Hello, World"
```

(note the double quotes) and see what happens. Look up the Lisp function `identity`. Congratulations! Only 751 operators to go, admittedly the more difficult 751, and you'll have implemented your own Common Lisp. If you have access to your implementation's source code, locate its definition for `identity` (if you're using SLIME see Chapter 18 for assistance).

Exercise

λλλ Functions can call themselves. *Recursion* is not necessarily the easiest way to set up a loop—we'll meet more satisfactory alternatives in Chapter 3—but feel free to get some practice in by implementing a function like `factorial` or `fibonacci`. What happens if you call your function with a non-numerical argument? With a numerical argument which isn't a whole number?

Beauty Tips, and Others

There are a few stylistic layout conventions which Lisp programmers follow. Please observe them rather than inventing your own. They'll make your code more readable both for you and for everybody else.

- Function definitions are laid out as in the example above, namely with `defun`, the function name and the argument list on one line, and the body of the function starting on the next line.
- "Correct" indentation is not compulsory—compare with Python—but you should act as if it is. It's a great aid to readability and is easily worth the minimal effort required to make it happen. Any decent Lisp-aware editor can indent code for you. Most such editors will indent the current line of Lisp code when you press the `TAB` key. Use it and learn from it. Some editors (e.g. `meta-control-q` in Emacs and LispWorks) also support indentation of several lines of code in one go, which is handy if you're not a great fan of RSI.
- If you have several parentheses together, keep them together: `)))))))` or whatever. Do not separate them from each other or from what precedes them with spaces or newlines.

A word about those parentheses. My definition of `factorial` ends with five consecutive close parentheses; a quick flit through the source code I maintain reveals several instances of ten or more consecutive close parentheses. This is not at all unusual. The

trick when you're working with parentheses is not to count them. Ever. When you're typing, your Lisp-aware editor will show you—typically with highlighting or colors—which open parenthesis each close corresponds to. Say you've decided to close the current defun. Press) repeatedly, watching on the screen for the (which is being closed. When the (in front of the `defun` lights up, you're done.

Mistakes are not unimaginable. Again, the editor is there to help you. Assuming you've indented code correctly (another good reason for *always* doing so), each successive open parenthesis which you match may or may not be above the previous one but should definitely be to the left of it. No? You've made a mistake. Select the line immediately below the open parenthesis which just matched. Press `TAB`. If the line jumps sideways, then the problem is either there or immediately above it. If not, keep working down.

I tend to write code from the outside in. So I might start by typing:

```
(defun frob (foo bar baz)
  )
```

I'll enter this "template", check that indentation and parentheses are correct, and then save the source file, all on autopilot. I might think for a while and enter some code:

```
(defun frob (foo bar baz)
  (if (frobbablep foo)
      ()                  ; what to do if I can frob foo
    ()))                  ; fallback in case I can't
```

Again, I've checked the indentation and parentheses. I try to keep the function looking like valid Lisp at all times (even if it doesn't do anything useful to start with).

Finally, here's how Lisp does comments:

- The semicolon `;` introduces a comment that lasts until the end of the current line.
- By convention, semicolon comments appended to the end of a line of code should be introduced by a single semicolon, whereas comments which get a line to themselves have two or more.
- To comment out several lines of code, start the comment with `#|` and terminate it with `|#`. If you use this to comment out further `#| ... |#` sets then these must be "balanced".

```
#|
;; This isn't very interesting. Anyway, Lisp defines a function called
;; max which does the same thing but with a variable number of arguments.
(defun greater-of-two (this that)
  (if (> this that)
      this                      ; this was bigger so we'll return it
    this))
|#
```

Strings and Characters

We've now met all three varieties of Lisp form.

- A *compound form* `(operator arguments ...)` has fixed evaluation rules when operator designates a function call and idiosyncratic ones otherwise. We've already covered function calls as well as `if` and `defun` and there'll be more macros and specials in Chapter 3.
- A *symbol*—such as the built-in constants `pi`, `nil` and `t`, or the arguments `this` and `that` to `(defun greater-of-two)` above—can be thought of in an evaluation context as representing a value. We'll revisit this in Chapter 3.
- Everything else is self-evaluating. This blanket statement accurately covers both the data types we've already met (the various numbers) and the many others to come later in the book.

Before I move on to Chapter 3 which deals with Lisp's control features in greater depth, I'd like to show you two more data types which will free our examples from the artificial restriction of only working with numbers. So let's take a brief look at strings and the characters of which they're comprised.

Character Syntax

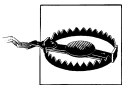
The Lisp character is a data type separate from numbers. All implementations must at the very least support ASCII's 95 graphic characters plus a newline character. In practice it won't stop there and the typical implementation nowadays supports the full Unicode range. Although I've thrown a Greek letter into a couple of the examples below for variety I'm not planning to go into any detail about anything beyond ASCII just yet; I will deal with this issue properly in Chapter 27.

Here are some characters:

```
#\a   #\Z   #\8   #\#   #\Space   #\Newline   #\λ
```

You'll see there are two syntax forms: `#\` followed by the character itself, or `#\` followed by the character's *name*. In source code the brief form will always work, provided your Lisp supports the character concerned and the format of your source file is capable of representing it. The long form will work for any character which has a name in your implementation (not all do).

Most of the details of how naming works are up to your implementation. It must use `#\Space` and `#\Newline` for the space and newline characters. If it supports certain other non-printing characters (in particular: `#\Tab`, `#\Return`, `#\Linefeed`) then it must name them thus.



Implementations are not obliged to name any other characters and when they do there is much variability. Be wary of hardwiring names other than `#\Space` and `#\Newline` into code which you intend to be portable across different implementations.

I already said that "everything else is self-evaluating", and so that's characters spoken for.

Exercise

λ Type some characters into the REPL.

This has been the fourth time we've met a syntax construct introduced the `#` character or *sharpsign*. The other three were hexadecimals, complex numbers and multiline comments. There are several others—a grand total of 22 altogether—and we'll meet just over half of them here. If you're curious to see the full set, take a look in section 2.4.8 in the *Syntax* chapter of the HyperSpec.

A Word About Newlines

No matter how many characters your operating system prefers to use for representing the end of a line of text, Lisp always uses precisely one character for this purpose: `#\Newline`. When reading from text files or writing back to them, an appropriate translation will take place to keep the OS happy.

Your implementation may choose to support the characters `#\Return` and `#\Linefeed`. If it does then it will probably provide some means of getting them into or out of text files individually; product documentation should supply the necessary details. Note that either (or conceivably both) of these characters may turn out to be identical to `#\Newline`:

```
#\Linefeed => #\Newline      ; in some implementations
```

Working with Characters

The predicate `char=` takes one or more characters and returns *`true`* if they are all the same. Similarly, `char/=` returns *`true`* if all the characters passed to it are different. If you want the comparison to be case-insensitive, use `char-equal` and `char-not-equal`.

```
(char= #\A #\A #\A)             => true
(char= #\B #\b #\b)             => NIL
(char= #\Linefeed #\Newline)    => true in some implementations
(char/= #\Linefeed #\Newline)   => true in some implementations
(char-equal #\B #\b #\b)        => true
```

Associated with each character is an integer, its *character code*. Theoretically, as with names, implementations have a free hand here. In practice all current Common Lisp implementations can be relied on to use ASCII encodings for the standard graphic characters.

```
(char-code #\+)    =>  43
(char-code #\a)    =>  97

(code-char 65)     =>  #\A
(code-char 97)     =>  #\a
```

Exercise

λλ Does your implementation support the "null character" (code `0`)? If so, does it have a name?

The ordering predicates `char<` and `char>` (and their case-insensitive equivalents `char-lessp` and `char-greaterp`) return results consistent with their arguments' character codes:

```
(char< #\e #\h #\l #\o)    => true, whether or not ASCII encodings are used
```

The predicate `alpha-char-p` (one argument) is true of alphabetic characters, similarly `digit-char-p` is true of digits and `alphanumericp` is true of either alphabetics or digits. `Upper-case-p` and `lower-case-p` are true of upper- and lowercase characters respectively; `both-case-p` is true of any character "with case" (which might not apply to all alphabetics—consider the Arabic alphabet, for instance).

Finally, `char-upcase` takes any character and returns the corresponding uppercase character if there is one, or the original argument if not; `char-downcase` works in the other direction.

Exercise

λλ Spend a few minutes experimenting with these functions.

Strings

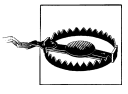
The syntax for a literal string is really straightforward. Strings are delimited by double quotes (the character `#\"`); the *single escape character* backslash `#\\` should precede any double quote or backslash which you wish to embed in the string. Strings can be multilined. That's it.

Lisp strings are not null-terminated. Lisp keeps track of every string's length and you can query it like this:

```
(length "Hello, World")      => 12
(length "\"")                => 1
(length "")                  => 0
```

You can access the characters which comprise a string with the *reader* `schar`. Note that Lisp strings—and the other array types, when we come to them—are all *zero indexed*. Also, unless you explicitly tell it not to, Lisp routinely checks that you haven't overshot the length of the string: no uncaught buffer overflows in Lisp!

```
(schar "Hello, World" 0)     => #\H
(schar "Hello, World" 12)    => an error
```



A string containing one character is *not* the *same thing* as that character. Certain Lisp functions are woolly in this regard: they'll accept a string of length one where a character was expected, or vice versa. I've never really approved of this feature and don't intend to dwell on it.

Exercise

λλ Write a function which returns the last character of a given string. What will you do if the string is empty?

You can extract substrings with the function `subseq`. This takes a string along with start and end indices. The result will *include* the character at position start and *exclude* the character at end:

```
(subseq "Hello, World" 3 8)  => "lo, W"
```

Or you can leave `end` out and the substring will continue to the end of the original:

```
(subseq "Hello, World" 7)    => "World"
```

There are several ways of splicing strings together, each with their own uses. Here's one: the function `format` which can be used more generally for splicing the printed representations of any Lisp objects into a template string. Think of `format` as C's `sprintf()` but much more powerful, almost a language within the language. This example is just the tip of the iceberg.

```
(format nil "~a~a number ~a" "Hi" #\, 42)  => "Hi, number 42"
```

The `nil` here means "return a string". Other values for this argument can be used to direct output to files, sockets, etc. We'll come back to `format` in Chapter 5.

Exercise

λλ Implement a "Hello World" function:

```
(hello "Nick")  => "Hello, Nick!"
```

All the character comparison predicates have string equivalents. In contrast with the character predicates, these functions compare precisely two strings at a time. (There's a reason for this and we'll come to it in Chapter 3.) Some examples:

```
(string= "Hello," "World")       => NIL
(string= "foo" "foo")            => true
(string-equal "Foo" "foo")       => true
(string< "lisper" "lispingly")   => true
(string< "hi" "Hitherto")        => NIL
(string-lessp "hi" "Hitherto")   => true
```

The functions `string-upcase`, `string-downcase` and `string-capitalize` each take and return a single string:

```
(string-upcase "they return strings.")      => "THEY RETURN STRINGS."
(string-capitalize "they return strings.")  => "They Return Strings."
```

Exercise

:λλλ A function which capitalized just the first word of a string and downcased the rest might also be useful. You can implement this in one line if you know the intricacies of `format`:

```
(defun string-capitalize-first (item)
  (format nil "~@(~a~)" item))
```

If you really enjoy writing recursive functions, see if you can get there using just the mechanisms introduced in this chapter.

```
(string-capitalize-first "-->start HERE<--")  => "-->Start here<--"
```

(Hint: This is much more work than it need be because I've hardly introduced any control features yet. Consider auxiliary functions. Alternatively, turn to Chapter 3.)