# 4. Syntax and Semantics

After that whirlwind tour, we'll settle down for a few chapters to take a more systematic look at the features you've used so far. I'll start with an overview of the basic elements of Lisp's syntax and semantics, which means, of course, that I must first address that burning question. . .

## What's with All the Parentheses?

Lisp's syntax is quite a bit different from the syntax of languages descended from Algol. The two most immediately obvious characteristics are the extensive use of parentheses and prefix notation. For whatever reason, a lot of folks are put off by this syntax. Lisp's detractors tend to describe the syntax as "weird" and "annoying." Lisp, they say, must stand for Lots of Irritating Superfluous Parentheses. Lisp folks, on the other hand, tend to consider Lisp's syntax one of its great virtues. How is it that what's so off-putting to one group is a source of delight to another?

I can't really make the complete case for Lisp's syntax until I've explained Lisp's macros a bit more thoroughly, but I can start with an historical tidbit that suggests it may be worth keeping an open mind: when John McCarthy first invented Lisp, he intended to implement a more Algol-like syntax, which he called *M-expressions*. However, he never got around to it. He explained why not in his article "History of Lisp."[1]

> The project of defining M-expressions precisely and compiling them or at least translating them into S-expressions was neither finalized nor explicitly abandoned. It just receded into the indefinite future, and a new generation of programmers appeared who preferred [S-expressions] to any FORTRAN-like or ALGOL-like notation that could be devised.

In other words, the people who have actually used Lisp over the past 45 years have *liked* the syntax and have found that it makes the language more powerful. In the next few chapters, you'll begin to see why.

## Breaking Open the Black Box

Before we look at the specifics of Lisp's syntax and semantics, it's worth taking a moment to look at how they're defined and how this differs from many other languages.

In most programming languages, the language processor--whether an interpreter or a compiler--operates as a black box: you shove a sequence of characters representing the text of a program into the black box, and it--depending on whether it's an interpreter or a compiler--either executes

the behaviors indicated or produces a compiled version of the program that will execute the behaviors when it's run.

Inside the black box, of course, language processors are usually divided into subsystems that are each responsible for one part of the task of translating a program text into behavior or object code. A typical division is to split the processor into three phases, each of which feeds into the next: a lexical analyzer breaks up the stream of characters into tokens and feeds them to a parser that builds a tree representing the expressions in the program, according to the language's grammar. This tree--called an *abstract syntax tree*--is then fed to an evaluator that either interprets it directly or compiles it into some other language such as machine code. Because the language processor is a black box, the data structures used by the processor, such as the tokens and abstract syntax trees, are of interest only to the language implementer.

In Common Lisp things are sliced up a bit differently, with consequences for both the implementer and for how the language is defined. Instead of a single black box that goes from text to program behavior in one step, Common Lisp defines *two* black boxes, one that translates text into Lisp objects and another that implements the semantics of the language in terms of those objects. The first box is called the *reader*, and the second is called the *evaluator*.[2]

Each black box defines one level of syntax. The reader defines how strings of characters can be translated into Lisp objects called *s-expressions*.[3] Since the s-expression syntax includes syntax for lists of arbitrary objects, including other lists, s-expressions can represent arbitrary tree expressions, much like the abstract syntax tree generated by the parsers for non-Lisp languages.

The evaluator then defines a syntax of Lisp *forms* that can be built out of s-expressions. Not all s-expressions are legal Lisp forms any more than all sequences of characters are legal s-expressions. For instance, both `(foo 1 2)` and `("foo" 1 2)` are s-expressions, but only the former can be a Lisp form since a list that starts with a string has no meaning as a Lisp form.

This split of the black box has a couple of consequences. One is that you can use s-expressions, as you saw in Chapter 3, as an externalizable data format for data other than source code, using **`READ`** to read it and **`PRINT`** to print it.[4] The other consequence is that since the semantics of the language are defined in terms of trees of objects rather than strings of characters, it's easier to generate code within the language than it would be if you had to generate code as text. Generating code completely from scratch is only marginally easier--building up lists vs. building up strings is about the same amount of work. The real win, however, is that you can generate code by manipulating existing data. This is the basis for Lisp's macros, which I'll discuss in much more detail in future chapters. For now I'll focus on the two levels of syntax defined by Common Lisp: the syntax of s-expressions understood by the reader and the syntax of Lisp forms understood by the evaluator.

# S-expressions

The basic elements of s-expressions are *lists* and *atoms*. Lists are delimited by parentheses and can contain any number of whitespace-separated elements. Atoms are everything else.[5] The elements of lists are themselves s-expressions (in other words, atoms or nested lists). Comments--which aren't, technically speaking, s-expressions--start with a semicolon, extend to the end of a line, and are treated essentially like whitespace.

And that's pretty much it. Since lists are syntactically so trivial, the only remaining syntactic rules you need to know are those governing the form of different kinds of atoms. In this section I'll describe the rules for the most commonly used kinds of atoms: numbers, strings, and names. After that, I'll cover how s-expressions composed of these elements can be evaluated as Lisp forms.

Numbers are fairly straightforward: any sequence of digits--possibly prefaced with a sign (`+` or `-`), containing a decimal point (`.`) or a solidus (`/`), or ending with an exponent marker--is read as a number. For example:

```
123       ; the integer one hundred twenty-three
3/7       ; the ratio three-sevenths
1.0       ; the floating-point number one in default precision
1.0e0     ; another way to write the same floating-point number
1.0d0     ; the floating-point number one in "double" precision
1.0e-4    ; the floating-point equivalent to one-ten-thousandth
+42       ; the integer forty-two
-42       ; the integer negative forty-two
-1/4      ; the ratio negative one-quarter
-2/8      ; another way to write negative one-quarter
246/2     ; another way to write the integer one hundred twenty-three
```

These different forms represent different kinds of numbers: integers, ratios, and floating point. Lisp also supports complex numbers, which have their own notation and which I'll discuss in Chapter 10.

As some of these examples suggest, you can notate the same number in many ways. But regardless of how you write them, all rationals--integers and ratios--are represented internally in "simplified" form. In other words, the objects that represent -2/8 or 246/2 aren't distinct from the objects that represent -1/4 and 123. Similarly, `1.0` and `1.0e0` are just different ways of writing the same number. On the other hand, `1.0`, `1.0d0`, and `1` can all denote different objects because the different floating-point representations and integers are different types. We'll save the details about the characteristics of different kinds of numbers for Chapter 10.

Strings literals, as you saw in the previous chapter, are enclosed in double quotes. Within a string a backslash (`\`) escapes the next character, causing it to be included in the string regardless of what it is. The only two characters that *must* be escaped within a string are double quotes and the backslash itself. All other characters can be included in a string literal without escaping, regardless of their meaning outside a string. Some example string literals are as follows:

```
"foo"     ; the string containing the characters f, o, and o.
"fo\o"    ; the same string
```

```
"fo\\o"   ; the string containing the characters f, o, \, and o.
"fo\"o"   ; the string containing the characters f, o, ", and o.
```

Names used in Lisp programs, such as **FORMAT** and `hello-world`, and `*db*` are represented by objects called *symbols*. The reader knows nothing about how a given name is going to be used--whether it's the name of a variable, a function, or something else. It just reads a sequence of characters and builds an object to represent the name.[6] Almost any character can appear in a name. Whitespace characters can't, though, because the elements of lists are separated by whitespace. Digits can appear in names as long as the name as a whole can't be interpreted as a number. Similarly, names can contain periods, but the reader can't read a name that consists only of periods. Ten characters that serve other syntactic purposes can't appear in names: open and close parentheses, double and single quotes, backtick, comma, colon, semicolon, backslash, and vertical bar. And even those characters *can*, if you're willing to escape them by preceding the character to be escaped with a backslash or by surrounding the part of the name containing characters that need escaping with vertical bars.

Two important characteristics of the way the reader translates names to symbol objects have to do with how it treats the case of letters in names and how it ensures that the same name is always read as the same symbol. While reading names, the reader converts all unescaped characters in a name to their uppercase equivalents. Thus, the reader will read `foo`, `Foo`, and `FOO` as the same symbol: `FOO`. However, `\f\o\o` and `|foo|` will both be read as `foo`, which is a different object than the symbol `FOO`. This is why when you define a function at the REPL and it prints the name of the function, it's been converted to uppercase. Standard style, these days, is to write code in all lowercase and let the reader change names to uppercase.[7]

To ensure that the same textual name is always read as the same symbol, the reader *interns* symbols--after it has read the name and converted it to all uppercase, the reader looks in a table called a *package* for an existing symbol with the same name. If it can't find one, it creates a new symbol and adds it to the table. Otherwise, it returns the symbol already in the table. Thus, anywhere the same name appears in any s-expression, the same object will be used to represent it.[8]

Because names can contain many more characters in Lisp than they can in Algol-derived languages, certain naming conventions are distinct to Lisp, such as the use of hyphenated names like `hello-world`. Another important convention is that global variables are given names that start and end with `*`. Similarly, constants are given names starting and ending in `+`. And some programmers will name particularly low-level functions with names that start with `%` or even `%%`. The names defined in the language standard use only the alphabetic characters (A-Z) plus `*`, `+`, `-`, `/`, `1`, `2`, `<`, `=`, `>`, and `&`.

The syntax for lists, numbers, strings, and symbols can describe a good percentage of Lisp programs. Other rules describe notations for literal vectors, individual characters, and arrays, which I'll cover when I talk about the associated data types in Chapters 10 and 11. For now the

key thing to understand is how you can combine numbers, strings, and symbols with parentheses-delimited lists to build s-expressions representing arbitrary trees of objects. Some simple examples look like this:

```
x             ; the symbol X
()            ; the empty list
(1 2 3)       ; a list of three numbers
("foo" "bar") ; a list of two strings
(x y z)       ; a list of three symbols
(x 1 "foo")   ; a list of a symbol, a number, and a string
(+ (* 2 3) 4) ; a list of a symbol, a list, and a number.
```

An only slightly more complex example is the following four-item list that contains two symbols, the empty list, and another list, itself containing two symbols and a string:

```
(defun hello-world ()
  (format t "hello, world"))
```

# S-expressions As Lisp Forms

After the reader has translated a bunch of text into s-expressions, the s-expressions can then be evaluated as Lisp code. Or some of them can--not every s-expressions that the reader can read can necessarily be evaluated as Lisp code. Common Lisp's evaluation rule defines a second level of syntax that determines which s-expressions can be treated as Lisp forms.[9] The syntactic rules at this level are quite simple. Any atom--any nonlist or the empty list--is a legal Lisp form as is any list that has a symbol as its first element.[10]

Of course, the interesting thing about Lisp forms isn't their syntax but how they're evaluated. For purposes of discussion, you can think of the evaluator as a function that takes as an argument a syntactically well-formed Lisp form and returns a value, which we can call the *value* of the form. Of course, when the evaluator is a compiler, this is a bit of a simplification--in that case, the evaluator is given an expression and generates code that will compute the appropriate value when it's run. But this simplification lets me describe the semantics of Common Lisp in terms of how the different kinds of Lisp forms are evaluated by this notional function.

The simplest Lisp forms, atoms, can be divided into two categories: symbols and everything else. A symbol, evaluated as a form, is considered the name of a variable and evaluates to the current value of the variable.[11] I'll discuss in Chapter 6 how variables get their values in the first place. You should also note that certain "variables" are that old oxymoron of programming: "constant variables." For instance, the symbol **`PI`** names a constant variable whose value is the best possible floating-point approximation to the mathematical constant *pi*.

All other atoms--numbers and strings are the kinds you've seen so far--are *self-evaluating* objects. This means when such an expression is passed to the notional evaluation function, it's simply returned. You saw examples of self-evaluating objects in Chapter 2 when you typed `10` and `"hello, world"` at the REPL.

It's also possible for symbols to be self-evaluating in the sense that the variables they name can be assigned the value of the symbol itself. Two important constants that are defined this way are **`T`** and **`NIL`**, the canonical true and false values. I'll discuss their role as booleans in the section "Truth, Falsehood, and Equality."

Another class of self-evaluating symbols are the *keyword* symbols--symbols whose names start with `:`. When the reader interns such a name, it automatically defines a constant variable with the name and with the symbol as the value.

Things get more interesting when we consider how lists are evaluated. All legal list forms start with a symbol, but three kinds of list forms are evaluated in three quite different ways. To determine what kind of form a given list is, the evaluator must determine whether the symbol that starts the list is the name of a function, a macro, or a special operator. If the symbol hasn't been defined yet--as may be the case if you're compiling code that contains references to functions that will be defined later--it's assumed to be a function name.[12] I'll refer to the three kinds of forms as *function call forms*, *macro forms*, and *special forms*.

# Function Calls

The evaluation rule for function call forms is simple: evaluate the remaining elements of the list as Lisp forms and pass the resulting values to the named function. This rule obviously places some additional syntactic constraints on a function call form: all the elements of the list after the first must themselves be well-formed Lisp forms. In other words, the basic syntax of a function call form is as follows, where each of the arguments is itself a Lisp form:

```
(function-name argument*)
```

Thus, the following expression is evaluated by first evaluating `1`, then evaluating `2`, and then passing the resulting values to the **`+`** function, which returns 3:

```
(+ 1 2)
```

A more complex expression such as the following is evaluated in similar fashion except that evaluating the arguments `(+ 1 2)` and `(- 3 4)` entails first evaluating their arguments and applying the appropriate functions to them:

```
(* (+ 1 2) (- 3 4))
```

Eventually, the values 3 and -1 are passed to the **`*`** function, which returns -3.

As these examples show, functions are used for many of the things that require special syntax in other languages. This helps keep Lisp's syntax regular.

# Special Operators

That said, not all operations can be defined as functions. Because all the arguments to a function are evaluated before the function is called, there's no way to write a function that behaves like the **IF** operator you used in Chapter 3. To see why, consider this form:

```
(if x (format t "yes") (format t "no"))
```

If **IF** were a function, the evaluator would evaluate the argument expressions from left to right. The symbol `x` would be evaluated as a variable yielding some value; then `(format t "yes")` would be evaluated as a function call, yielding **NIL** after printing "yes" to standard output. Then `(format t "no")` would be evaluated, printing "no" and also yielding **NIL**. Only after all three expressions were evaluated would the resulting values be passed to **IF**, too late for it to control which of the two **FORMAT** expressions gets evaluated.

To solve this problem, Common Lisp defines a couple dozen so-called special operators, **IF** being one, that do things that functions can't do. There are 25 in all, but only a small handful are used directly in day-to-day programming.[13]

When the first element of a list is a symbol naming a special operator, the rest of the expressions are evaluated according to the rule for that operator.

The rule for **IF** is pretty easy: evaluate the first expression. If it evaluates to non-**NIL**, then evaluate the next expression and return its value. Otherwise, return the value of evaluating the third expression or **NIL** if the third expression is omitted. In other words, the basic form of an **IF** expression is as follows:

```
(if test-form then-form [ else-form ])
```

The *test-form* will always be evaluated and then one or the other of the *then-form* or *else-form*.

An even simpler special operator is **QUOTE**, which takes a single expression as its "argument" and simply returns it, unevaluated. For instance, the following evaluates to the list `(+ 1 2)`, not the value 3:

```
(quote (+ 1 2))
```

There's nothing special about this list; you can manipulate it just like any list you could create with the **LIST** function.[14]

**QUOTE** is used commonly enough that a special syntax for it is built into the reader. Instead of writing the following:

```
(quote (+ 1 2))
```

you can write this:

```
'(+ 1 2)
```

This syntax is a small extension of the s-expression syntax understood by the reader. From the point of view of the evaluator, both those expressions will look the same: a list whose first element is the symbol **QUOTE** and whose second element is the list `(+ 1 2)`.[15]

In general, the special operators implement features of the language that require some special processing by the evaluator. For instance, several special operators manipulate the environment in which other forms will be evaluated. One of these, which I'll discuss in detail in Chapter 6, is **LET**, which is used to create new variable bindings. The following form evaluates to 10 because the second `x` is evaluated in an environment where it's the name of a variable established by the **LET** with the value 10:

```
(let ((x 10)) x)
```

# Macros

While special operators extend the syntax of Common Lisp beyond what can be expressed with just function calls, the set of special operators is fixed by the language standard. Macros, on the other hand, give users of the language a way to extend its syntax. As you saw in Chapter 3, a macro is a function that takes s-expressions as arguments and returns a Lisp form that's then evaluated in place of the macro form. The evaluation of a macro form proceeds in two phases: First, the elements of the macro form are passed, unevaluated, to the macro function. Second, the form returned by the macro function--called its *expansion*--is evaluated according to the normal evaluation rules.

It's important to keep the two phases of evaluating a macro form clear in your mind. It's easy to lose track when you're typing expressions at the REPL because the two phases happen one after another and the value of the second phase is immediately returned. But when Lisp code is compiled, the two phases happen at completely different times, so it's important to keep clear what's happening when. For instance, when you compile a whole file of source code with the function **COMPILE-FILE**, all the macro forms in the file are recursively expanded until the code consists of nothing but function call forms and special forms. This macroless code is then compiled into a FASL file that the **LOAD** function knows how to load. The compiled code, however, isn't executed until the file is loaded. Because macros generate their expansion at compile time, they can do relatively large amounts of work generating their expansion without having to pay for it when the file is loaded or the functions defined in the file are called.

Since the evaluator doesn't evaluate the elements of the macro form before passing them to the macro function, they don't need to be well-formed Lisp forms. Each macro assigns a meaning to the s-expressions in the macro form by virtue of how it uses them to generate its expansion. In other words, each macro defines its own local syntax. For instance, the `backwards` macro from Chapter 3 defines a syntax in which an expression is a legal `backwards` form if it's a list that's the reverse of a legal Lisp form.

I'll talk quite a bit more about macros throughout this book. For now the important thing for you to realize is that macros--while syntactically similar to function calls--serve quite a different purpose, providing a hook into the compiler.[16]

# Truth, Falsehood, and Equality

Two last bits of basic knowledge you need to get under your belt are Common Lisp's notion of truth and falsehood and what it means for two Lisp objects to be "equal." Truth and falsehood are--in this realm--straightforward: the symbol **NIL** is the only false value, and everything else is true. The symbol **T** is the canonical true value and can be used when you need to return a non-**NIL** value and don't have anything else handy. The only tricky thing about **NIL** is that it's the only object that's both an atom and a list: in addition to falsehood, it's also used to represent the empty list.[17] This equivalence between **NIL** and the empty list is built into the reader: if the reader sees `()`, it reads it as the symbol **NIL**. They're completely interchangeable. And because **NIL**, as I mentioned previously, is the name of a constant variable with the symbol **NIL** as its value, the expressions `nil`, `()`, `'nil`, and `'()` all evaluate to the same thing--the unquoted forms are evaluated as a reference to the constant variable whose value is the symbol **NIL**, but in the quoted forms the **QUOTE** special operator evaluates to the symbol directly. For the same reason, both `t` and `'t` will evaluate to the same thing: the symbol **T**.

Using phrases such as "the same thing" of course begs the question of what it means for two values to be "the same." As you'll see in future chapters, Common Lisp provides a number of type-specific equality predicates: **=** is used to compare numbers, **CHAR=** to compare characters, and so on. In this section I'll discuss the four "generic" equality predicates--functions that can be passed any two Lisp objects and will return true if they're equivalent and false otherwise. They are, in order of discrimination, **EQ**, **EQL**, **EQUAL**, and **EQUALP**.

**EQ** tests for "object identity"--two objects are **EQ** if they're identical. Unfortunately, the object identity of numbers and characters depends on how those data types are implemented in a particular Lisp. Thus, **EQ** may consider two numbers or two characters with the same value to be equivalent, or it may not. Implementations have enough leeway that the expression `(eq 3 3)` can legally evaluate to either true or false. More to the point, `(eq x x)` can evaluate to either true or false if the value of `x` happens to be a number or character.

Thus, you should never use **EQ** to compare values that may be numbers or characters. It may seem to work in a predictable way for certain values in a particular implementation, but you have no guarantee that it will work the same way if you switch implementations. And switching implementations may mean simply upgrading your implementation to a new version--if your Lisp implementer changes how they represent numbers or characters, the behavior of **EQ** could very well change as well.

Thus, Common Lisp defines **EQL** to behave like **EQ** except that it also is guaranteed to consider two objects of the same class representing the same numeric or character value to be equivalent. Thus, `(eql 1 1)` is guaranteed to be true. And `(eql 1 1.0)` is guaranteed to be false since the integer value 1 and the floating-point value are instances of different classes.

There are two schools of thought about when to use **EQ** and when to use **EQL**: The "use **EQ** when possible" camp argues you should use **EQ** when you know you aren't going to be com-paring numbers or characters because (a) it's a way to indicate that you aren't going to be comparing numbers or characters and (b) it will be marginally more efficient since **EQ** doesn't have to check whether its arguments are numbers or characters.

The "always use **EQL**" camp says you should never use **EQ** because (a) the potential gain in clarity is lost because every time someone reading your code--including you--sees an **EQ**, they have to stop and check whether it's being used correctly (in other words, that it's never going to be called upon to compare numbers or characters) and (b) that the efficiency difference between **EQ** and **EQL** is in the noise compared to real performance bottlenecks.

The code in this book is written in the "always use **EQL**" style.[18]

The other two equality predicates, **EQUAL** and **EQUALP**, are general in the sense that they can operate on all types of objects, but they're much less fundamental than **EQ** or **EQL**. They each define a slightly less discriminating notion of equivalence than **EQL**, allowing different objects to be considered equivalent. There's nothing special about the particular notions of equivalence these functions implement except that they've been found to be handy by Lisp programmers in the past. If these predicates don't suit your needs, you can always define your own predicate function that compares different types of objects in the way you need.

**EQUAL** loosens the discrimination of **EQL** to consider lists equivalent if they have the same structure and contents, recursively, according to **EQUAL**. **EQUAL** also considers strings equivalent if they contain the same characters. It also defines a looser definition of equivalence than **EQL** for bit vectors and pathnames, two data types I'll discuss in future chapters. For all other types, it falls back on **EQL**.

**EQUALP** is similar to **EQUAL** except it's even less discriminating. It considers two strings equivalent if they contain the same characters, ignoring differences in case. It also considers two characters equivalent if they differ only in case. Numbers are equivalent under **EQUALP** if they represent the same mathematical value. Thus, `(equalp 1 1.0)` is true. Lists with **EQUALP** elements are **EQUALP**; likewise, arrays with **EQUALP** elements are **EQUALP**. As with **EQUAL**, there are a few other data types that I haven't covered yet for which **EQUALP** can consider two objects equivalent that neither **EQL** nor **EQUAL** will. For all other data types, **EQUALP** falls back on **EQL**.

# Formatting Lisp Code

While code formatting is, strictly speaking, neither a syntactic nor a semantic matter, proper formatting is important to reading and writing code fluently and idiomatically. The key to formatting Lisp code is to indent it properly. The indentation should reflect the structure of the code so that you don't need to count parentheses to see what goes with what. In general, each new level of nesting gets indented a bit more, and, if line breaks are necessary, items at the same level of nesting are lined up. Thus, a function call that needs to be broken up across multiple lines might be written like this:

```
(some-function arg-with-a-long-name
               another-arg-with-an-even-longer-name)
```

Macro and special forms that implement control constructs are typically indented a little differently: the "body" elements are indented two spaces relative to the opening parenthesis of the form. Thus:

```
(defun print-list (list)
  (dolist (i list)
    (format t "item: ~a~%" i)))
```

However, you don't need to worry too much about these rules because a proper Lisp environment such as SLIME will take care of it for you. In fact, one of the advantages of Lisp's regular syntax is that it's fairly easy for software such as editors to know how to indent it. Since the indentation is supposed to reflect the structure of the code and the structure is marked by parentheses, it's easy to let the editor indent your code for you.

In SLIME, hitting Tab at the beginning of each line will cause it to be indented appropriately, or you can re-indent a whole expression by positioning the cursor on the opening parenthesis and typing `C-M-q`. Or you can re-indent the whole body of a function from anywhere within it by typing `C-c M-q`.

Indeed, experienced Lisp programmers tend to rely on their editor handling indenting automatically, not just to make their code look nice but to detect typos: once you get used to how code is supposed to be indented, a misplaced parenthesis will be instantly recognizable by the weird indentation your editor gives you. For example, suppose you were writing a function that was supposed to look like this:

```
(defun foo ()
  (if (test)
    (do-one-thing)
    (do-another-thing)))
```

Now suppose you accidentally left off the closing parenthesis after `test`. Because you don't bother counting parentheses, you quite likely would have added an extra parenthesis at the end of the **`DEFUN`** form, giving you this code:

```
(defun foo ()
  (if (test
    (do-one-thing)
    (do-another-thing))))
```

However, if you had been indenting by hitting Tab at the beginning of each line, you wouldn't have code like that. Instead you'd have this:

```
(defun foo ()
  (if (test
       (do-one-thing)
       (do-another-thing))))
```

Seeing the then and else clauses indented way out under the condition rather than just indented slightly relative to the **`IF`** shows you immediately that something is awry.

Another important formatting rule is that closing parentheses are always put on the same line as the last element of the list they're closing. That is, don't write this:

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)
  )
)
```

but instead write this:

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)))
```

The string of ) ) ) s at the end may seem forbidding, but as long your code is properly indented the parentheses should fade away--no need to give them undue prominence by spreading them across several lines.

Finally, comments should be prefaced with one to four semicolons depending on the scope of the comment as follows:

```
;;;; Four semicolons are used for a file header comment.

;;; A comment with three semicolons will usually be a paragraph
;;; comment that applies to a large section of code that follows,

(defun foo (x)
  (dotimes (i x)
    ;; Two semicolons indicate this comment applies to the code
    ;; that follows. Note that this comment is indented the same
    ;; as the code that follows.
    (some-function-call)
    (another i)              ; this comment applies to this line only
    (and-another)            ; and this is for this line
    (baz)))
```

Now you're ready to start looking in greater detail at the major building blocks of Lisp programs, functions, variables, and macros. Up next: functions.

---

[1] http://www-formal.stanford.edu/jmc/history/lisp/node3.html

[2]Lisp implementers, like implementers of any language, have many ways they can implement an evaluator, ranging from a "pure" interpreter that interprets the objects given to the evaluator directly to a compiler that translates the objects into machine code that it then runs. In the middle are implementations that compile the input into an intermediate form such as bytecodes for a virtual machine and then interprets the bytecodes. Most Common Lisp implementations these days use some form of compilation even when evaluating code at run time.

[3]Sometimes the phrase *s-expression* refers to the textual representation and sometimes to the objects that result from reading the textual representation. Usually either it's clear from context which is meant or the distinction isn't that important.

[4]Not all Lisp objects can be written out in a way that can be read back in. But anything you can **`READ`** can be printed back out "readably" with **`PRINT`**.

[5]The empty list, `()`, which can also be written **`NIL`**, is both an atom and a list.

[6]In fact, as you'll see later, names aren't intrinsically tied to any one kind of thing. You can use the same name, depending on context, to refer to both a variable and a function, not to mention several other possibilities.

[7]The case-converting behavior of the reader can, in fact, be customized, but understanding when and how to change it requires a much deeper discussion of the relation between names, symbols, and other program elements than I'm ready to get into just yet.

[8]I'll discuss the relation between symbols and packages in more detail in Chapter 21.

[9]Of course, other levels of correctness exist in Lisp, as in other languages. For instance, the s-expression that results from reading `(foo 1 2)` is syntactically well-formed but can be evaluated only if `foo` is the name of a function or macro.

[10]One other rarely used kind of Lisp form is a list whose first element is a *lambda form*. I'll discuss this kind of form in Chapter 5.

[11]One other possibility exists--it's possible to define *symbol macros* that are evaluated slightly differently. We won't worry about them.

[12]In Common Lisp a symbol can name both an operator--function, macro, or special operator--and a variable. This is one of the major differences between Common Lisp and Scheme. The difference is sometimes described as Common Lisp being a Lisp-2 vs. Scheme being a Lisp-1--a Lisp-2 has two namespaces, one for operators and one for variables, but a Lisp-1 uses a single namespace. Both choices have advantages, and partisans can debate endlessly which is better.

[13]The others provide useful, but somewhat esoteric, features. I'll discuss them as the features they support come up.

[14]Well, one difference exists--literal objects such as quoted lists, but also including double-quoted strings, literal arrays, and vectors (whose syntax you'll see later), must not be modified. Consequently, any lists you plan to manipulate you should create with **`LIST`**.

[15]This syntax is an example of a *reader macro*. Reader macros modify the syntax the reader uses to translate text into Lisp objects. It is, in fact, possible to define your own reader macros, but that's a rarely used facility of the language. When most Lispers talk about "extending the syntax" of the language, they're talking about regular macros, as I'll discuss in a moment.

[16]People without experience using Lisp's macros or, worse yet, bearing the scars of C preprocessor-inflicted wounds, tend to get nervous when they realize that macro calls look like regular function calls. This turns out not to be a problem in practice for several reasons. One is that macro forms are usually formatted differently than function calls. For instance, you write the following:

```
(dolist (x foo)
  (print x))
```

rather than this:

```
(dolist (x foo) (print x))
```

or

```
(dolist (x foo)
        (print x))
```

the way you would if **DOLIST** was a function. A good Lisp environment will automatically format macro calls correctly, even for user-defined macros.

And even if a **DOLIST** form was written on a single line, there are several clues that it's a macro: For one, the expression `(x foo)` is meaningful by itself only if `x` is the name of a function or macro. Combine that with the later occurrence of `x` as a variable, and it's pretty suggestive that **DOLIST** is a macro that's creating a binding for a variable named `x`. Naming conventions also help--looping constructs, which are invariably macros--are frequently given names starting with *do*.

[17]Using the empty list as false is a reflection of Lisp's heritage as a list-processing language much as the use of the integer 0 as false in C is a reflection of its heritage as a bit-twiddling language. Not all Lisps handle boolean values the same way. Another of the many subtle differences upon which a good Common Lisp vs. Scheme flame war can rage for days is Scheme's use of a distinct false value `#f`, which isn't the same value as either the symbol `nil` or the empty list, which are also distinct from each other.

[18]Even the language standard is a bit ambivalent about which of **EQ** or **EQL** should be preferred. *Object identity* is defined by **EQ**, but the standard defines the phrase *the same* when talking about objects to mean **EQL** unless another predicate is explicitly mentioned. Thus, if you want to be 100 percent technically correct, you can say that `(- 3 2)` and `(- 4 3)` evaluate to "the same" object but not that they evaluate to "identical" objects. This is, admittedly, a bit of an *angels-on-pinheads* kind of issue.