Common Lisp by Example

A compilation of notes from various sources

Preface

Lisp was invented in 1958 by John McCarthy and is the second oldest programming language in active use after FORTRAN. It is an elegant, industrial-strength language that has stood the test of time for over fifty years and remains one of the most powerful languages today. In 1994, the American National Standards Institute published the ANSI Common Lisp specification, under the guidance of renowned language designers Scott Fahlman, Richard Gabriel, David Moon, Kent Pitman, Guy Steele and Dan Weinreb. ANSI Common Lisp is the subject of this guide.

At a most basic level, a programming language is a set of rules (syntax) to convert text into machine code. Source code is stored as strings in most text-based programming languages and must be parsed by the compiler according to the syntax rules of the language. This is not the case for Lisp, which expresses source codes in a primitive data structure known as lists that can be directly evaluated by a Lisp compiler or interpreter.

This has a profound impact on the use of Lisp and the benefits it confers to its users. Lisp code is a first-class data object in the language, and can be manipulated in much the same way as any other data structure such as being dynamically created, destroyed, passed to a function, returned as a value, and have all the rights that other variables in the programming language have. This allows us to write programs that can generate and manipulate source code in much the same way as any other data object. In this sense we say that Lisp programs can write their own program, a truly remarkable result that resembles the next frontier of automation.

The World's First Functional Programming Language

Whilst Common Lisp is a powerful general-purpose, multi paradigm programming language, its roots lie in functional programming. In fact, John McCarthy's original Lisp was the world's first functional programming language. Functional programming is centred on defining and then utilising functions to generate programs. Simple functions are combined to build more complicated ones, the result of each is passed as the argument to the next, and the result of the last one is the result of the whole.

Side-effects represent changes to the state of the world as a result of evaluating an expression. Assignment of a value to a variable is the most common side effect. Printing to a screen or file is another common example. When we write code without side-effects, there is no point in defining functions with bodies of more than one expression. The value of the last expression in a function is the value returned by it and the values of all preceding expressions are discarded. If these preceding expressions did not have any side-effects, we would be unable to determine whether they evaluated at all. Functions with more than one expression likely imply the existence and use of side-effects.

Functions without side-effects can be easier to debug as there is a more direct transformation from input to output and we do not need to track changes in the values of variables and how they impact a function's output (noting that if multiple functions can change the same variable, we need to track all their changes to understand how the variables affect the function's output). That said, there are many cases where side-effects are useful and help in writing clearer code.

Indeed, each programming paradigm, whether it be functional, procedural, object-oriented or another paradigm, is well suited for certain problems and ill-suited for others. Fortunately Lisp is multi-paradigm and can be flexibly applied to many problem sets. In fact, many of the language features of Lisp have been adopted by other languages over the last thirty years and it remains one of the most powerful and flexible languages available today.

We will refrain from discussing functional programming further in this guide, except for the remark that it is useful to learn Lisp, at least initially, from a functional perspective. Practically this means we will be primarily focused with the values returned by functions and how the outputs of one function are used as inputs to another, both as part of a composite function that we arrange in our code. In fact, every Lisp expression can be thought of as a function that returns a value and, optionally, performs some side-effect.

Credits & Disclaimer

This guide builds on leading books on Lisp, particularly Common Lisp the Language, 2nd Edition by Guy L. Steele, ANSI Common Lisp by Paul Graham, Common Lisp: A Gentle Introduction to Symbolic Computation by David S. Touretzky and Practical Common Lisp by Peter Seibel. Credit must also be given to the StackOverflow community for their helpful answers over the years, in particular Rainer Joswig, Sylwester, coredump and tfb. Finally, comments and suggestions by the Reddit R/Lisp community have helped shape this guide, including ramenbytes for his assistance on macros). This guide is for educational purposes only and constitutes a 'fair use' of any copyrighted material (referenced and provided for in section 107 of the US Copyright Law).

Table of Contents

1. Introduction

Lisp is an interactive language and all Lisp systems will include an interactive front-end known as the **toplevel**. We can enter Lisp expressions into the toplevel and Lisp will evaluate them and print their value. This is known as the Read-Eval-Print-Loop, or **REPL** for short. Lisp expressions that are meant to be evaluated are known as **forms**.

Atoms are the most basic unit of Lisp, representing singular objects, such as the number 5 and the string "Hello, World!". **Lists** are collections of atoms or other lists, separated by whitespace and enclosed in parenthesis, such as (1 2 3) or ("Michael" "David" "Ben"). An example of a nested list would be ((1 2 3) 4 5 6).

Lisp utilises prefix notation throughout the language, meaning operators precede their arguments. As an example, we would calculate 2 + 3 as (+ 2 3).

1.1 Evaluation of Forms

Lisp applies a consistent set of evaluation rules for atomic and list forms. It is very important to have a deep understanding of Lisp's evaluation rules and this is the most important section of this guide.

Atoms evaluate to themselves. For example, the number 8 appearing as an atom within Lisp code will evaluate to 8 and the string "Color is gold" will evaluate to itself. This rule also applies to atomic symbols (discussed shortly), which will evaluate to the value bound to the symbol or return an error if the value is unbound. Note that lists themselves are a collection of atoms; in evaluating a list, we need to evaluate each of the atoms individually and then apply the rules for the evaluation of the list.

Lisp will evaluate a list by looking up its first element and determining whether it represents the start of one of four types of forms: function, special, macro or lambda. A list whose first element is the symbol for a function name is known as a **function form**. Lisp will evaluate the remaining elements of a function form and pass them as arguments to the function bound to (i.e. named by) the symbol. For example, the below form evaluates the arguments to 10, 5 and 15 respectively (each itself a result of a function), and pass them to the function call to return 30:

```
(+ (+ 5 5) (* 5 1) (+ 10 5))                    ; Returns 30
```

The evaluation rule for functions is not suitable for all scenarios. For example, in an IF conditional, we only want the true statement to be evaluated if the condition is true and we only want the false statement to be evaluated if the condition is false. For example, in the below, we only want to divide 10 by the value stored in X if it is not zero (as otherwise we will get divide-by-zero error). In an IF conditional, and in many other language constructs, we do not want each element of the list to be evaluated.

```
(defparameter x 0)
(if (zerop X)                          ; Test if x = 0
    "Cannot divide as X is zero"       ; Do this if x = 0
    (/ 10 X))                          ; Do this if x not 0
```

Common Lisp has 25 **special forms**, representing primitive functions with their own specific evaluation rules. The evaluation rule for the IF special form is to evaluate the true statement only if the condition is true and the false statement only if the condition is false. We will discuss the evaluation rules for common special forms in the remainder of this guide.

Finally, Lisp also has specific evaluation rules for **macro** and **lambda forms**. We will discuss these forms in greater depth towards the end of this guide. During this guide however we will sporadically introduce some of Common Lisp's built-in macros. For now, we can treat them similarly to special forms, noting each macro has a specific evaluation rule.

Try entering the following list into our toplevel. You will receive an error. This is because Lisp will look at the first element 1 and is unable to evaluate it as one of the four compound forms we discussed as 1 is a number, not the name of a function, special form or macro:

```
(1 2 3)
```

1.2 Controlling Evaluation

The quote special form is used to override the above evaluation rules by returning back the object literally without evaluation. For example, either of the below two forms will evaluate without error as Lisp is no longer evaluating the list (1 2 3), but rather returning it literally:

```
(quote (1 2 3))

'(1 2 3)                    ; ' is shorthand for quote
```

2. Symbols & Data Objects

We mentioned symbols in passing earlier but it is worth expanding on them a bit more here. Symbols are fully fledged objects in Lisp and we will discuss them in depth later. For our current purposes however we will focus on their role as names for objects. All variables and functions are named by a symbol, and so are special forms and macros.

When we enter any non-numeric code that is not a quoted string in Lisp, we are generally entering in a symbol. Lisp will then look up the symbol in a global table and retrieve the relevant object for evaluation. You can start to see the consistency of the language. Lisp extracts the first element of each form it receives, and looks up the symbol named by this element to get the appropriate evaluation rule. It then applies this evaluation rule to the form.

2.1 Variables

Lisp has a unique approach to variables and data objects. Variables are a place in computer memory that hold a pointer to a data object. Assigning a variable a new value will change what object the variable points to, but has no effect on the previously referenced object. That said, if a variable holds a reference to a mutable object, we can use that reference to modify that object and the modification will be visible to any code that also has a reference to the same object.

Variables themselves do not have a type, it is the underlying data object that is typed. A variable can have any Lisp object as its value. Lisp is accordingly a dynamically-typed language.

3. Global Variables & Assignment

DEFPARAMETER and DEFVAR establish global variables, known more technically as special or dynamic variables. These variables can be accessed throughout a Lisp program.

If a local variable exists with the same name as a global variable, a lexical closure will typically refer to the local variable. To avoid confusion, we should always name special variables with leading and trailing asterisks *, such as in the below examples.

DEFPARAMETER will unconditionally assign the value supplied to the variable, while DEFVAR will only assign the value if it is not already bound to the variable. A value must

be supplied to the DEFPARAMETER macro, whilst DEFVAR can establish a dynamic variable without assigning it a value:

```
(defvar *x*)              ; Establish an unbound variable

(defparameter *x* 15)     ; Assign the value 15 to X

(defvar *x* 10)           ; Does nothing as X already bound
```

We can define global constants with DEFCONSTANT. Note that global constants cannot be used as function parameters or rebound to a new value at a later stage.

```
(defconstant +my-constant+ 20)
```

3.1 General Purpose Assignment

SETF is Lisp's general purpose assignment macro that assigns a value to a designated place (such as a variable). It is a very powerful setter that can assign values to many different objects.

We won't go into detail, but rather provide illustrative examples throughout this guide. The syntax of SETF is as follows:

```
(setf place value)
```

Below are some examples.

```
(setf x 10)                    ; Set x to 10

(setf x 1 y 2)                 ; Set x to 1 and y to 2

(incf x)                       ; Same as (setf x (+ x 1))

(decf x)                       ; Same as (setf x (- x 1))

(incf x 10)                    ; Same as (setf x (+ x 10))
```

4. Input & Output

The second important side-effect after assignment is input & output. To write meaningful programs we need to be able to accept user input and provide output to the user. In this section, we cover the basic functions to read & print to a screen. Later in the guide we will cover the useful functions of reading & writing to a file.

4.1 Lisp Reader

The READ function takes characters from a stream, interprets them as a printed representation of a Lisp object, builds a Lisp object and finally returns it as the value of the function. The collection of routines that does this is called the (Lisp) reader. There are three primary variations of the Lisp reader:

- The function READ is used to parse input into Lisp objects and reads exactly one expression, regardless of newlines
- The function READ-LINE reads all characters up to a newline, returning a string
- The function READ-FROM-STRING takes a string and returns the first expression from it

Below are some examples you can experiment with.

```
(defparameter my-variable nil)

(setf my-variable (read-line))

(setf my-variable (read))

(setf my-variable (read-from-string))
```

4.2 Lisp Printer

Functions such as PRINT take a Lisp object and send the characters of its printed representation to a stream. The collection of routines that does this is known as the (Lisp) printer. PRIN1 and PRINT are used to generate output useful for programs (the latter adds a newline to the output whilst the former does not). PRINC is used for output for people. TERPRI prints a newline.

The FORMAT function is the most commonly used printer in Lisp. Its syntax is as follows.

```
(format destination control-string optional-arguments*)
```

The first argument of the FORMAT function is the destination where the output will be printed. A value of T will send the out to the stream *standard-output* (typically the main screen of your Lisp system) whilst NIL here will return the output as a string. We can alternatively supply a stream pointing to a file in our file system, and this is how we use FORMAT to write to files (we will discuss this later in this guide).

The second argument of the FORMAT function is the string that we want to print. However, we can enter directives (preceded by ~) to add complex behaviour to the string, such as printing newlines (~%) (i.e. TERPRI) or for inserting the printed representation of Lisp objects into the string (~A to print as PRINC, i.e. output for people and ~S to print as PRIN1, i.e. output suitable as input for the READ function).

The third (optional) argument of the FORMAT function is the arguments we want to supply to the control string. Each ~A or ~S in the control-string takes a successive argument from here and places it into the string.

This is best illustrated by the following examples. Note how Bob is quoted in the second example as the printed representation of Lisp strings includes quotes.

```
;; Prints to screen:
Dear Bob,
 How are you?

(format t "Dear ~A, ~% How are you?" "Bob")

;; Prints to screen:
Dear "Bob", How are you?

(format t "Dear ~S, How are you?" "Bob")

;; Returns "Number is: 3" (a string)

(format nil "~A ~A" "Number is:" (+ 1 2))
```

5. Global Functions

In our earlier examples, we used some of Lisp's built-in functions such as + (for addition), READ (for reading input) and FORMAT (for printing input). We will discuss how to define our own functions and also introduce some common functions for manipulating numbers and text.

Global functions are defined with DEFUN with the below syntax. The * implies that multiple forms can be included in these parts. body-form denotes the body of the function, which is wrapped in an implicit BLOCK with the same name as function-name. Functions evaluate the expressions of their bodies in order and return the value of the last expression, unless exited earlier with RETURN-FROM. We will discuss BLOCK and RETURN-FROM in the next section.

```
(defun function-name (parameter*)
  "Optional documentation string."
  body-form*)
```

An example of a function that multiplies the sum of two numbers by 10 is below, together with its function call.

```
(defun multiply-sum-by-10 (x y)
  "Returns the sum of two numbers multiplied by 10"
  (* 10 (+ x y)))

(multiply-sum-by-10 5 10)                    ; Returns 150
```

5.1 Numerical Functions

Some of Lisp's most basic functions are for arithmetic, e.g. (+ 3 3.0) for addition, (- 3 3.0) for subtraction, (* 3 3.0) for multiplication and (/ 3 3.0) for division. Numerical functions disregard type, i.e. 3 and 3.0 are treated as equal and we can complete arithmetic operations on disparate numerical types such as integer and floating point numbers without error. We can compare numbers with the below functions. With two arguments, they perform the usual arithmetic comparison tests. With three or more arguments, they are useful for range checks.

=	/= (not)	<	>	<=	>=

Below are examples of these functions.

```
(defparameter x 5)

(defparameter y 4)

(<= 0 x 5)      ; true as x between 0 and 5 inclusive

(< 0 x 5)       ; false as x not between 0 and 5 exclusive

(< 0 x y 6)     ; false as x > y

(< 0 y x 6)     ; true
```

Two other useful functions are MAX and MIN. Finally, we can use ABS to return the absolute value of one number.

5.2 Text Functions

We will now cover some useful functions for joining strings (CONCATENATE), retrieving their length (LENGTH), extracting a portion of a string (SUBSEQ) and searching within a string (SEARCH). The FORMAT function that we discussed earlier is very useful for constructing strings made up of smaller strings that we supply into the FORMAT function as arguments.

Below are examples of joining and searching within a string and retrieving its length in characters. SEARCH will return the starting point of the term searched or NIL if not found.

```
(concatenate 'string "Hello, " "world" ". Today is good.")

(length "Common")                          ; Returns 6

(search "term" "the term is search within this string")
```

SUBSEQ retrieves a portion of a string starting from a supplied starting position (indexed from 0) and an optional ending position (which is not included in the string):

```
(subseq "Common Lisp" 7 11)                ; Returns "Lisp"
```

The below table outlines comparison functions for strings. Replace STRING with CHAR in the below to get the equivalent character comparison functions.

Case Sensitive	**Case Insensitive**

`STRING=` `STRING<` `STRING>` `STRING<=` `STRING>=` `STRING/=`
 	
Note on usage: A string A is less than a string B if in the first position in which they differ the character of A is less than the corresponding character of B according to the function CHAR<, or if string a is a proper prefix of string B (of shorter length and matching in all the characters of A).
	`STRING-EQUAL`
	`STRING-NOT-EQUAL`
	
	`STRING-LESSP`
	`STRING-NOT-LESSP`
	
	`STRING-GREATERP`
	`STRING-NOT-GREATERP`

6. Blocks

Until now we have been working with singular Lisp forms. Frequently however, we want to evaluate a block of code together, such as within functions, or within an if statement or a loop. The PROGN special operator allows us to evaluate a sequence of Lisp forms in order and return the value of the last as the value of the PROGN form. Below is an example:

```
(progn
  (print "Hello")
  (print "World")
  (+ 5 5))                                ; Returns 10
```

The BLOCK special operator is similar, but it is named and has a mechanism for out-of-order exit with the RETURN-FROM operator. As mentioned earlier, the bodies of functions are wrapped in an implicit BLOCK.

```
(block my-block
  (print "We see this")
  (return-from my-block 10)               ; Returns 10
  (print "We will never see this"))
```

The RETURN macro returns its argument as the value of an enclosing BLOCK named NIL.

Many Common Lisp operators that take a body of expressions implicitly enclose the body in a BLOCK named NIL and we can use RETURN in these forms:

```
(dolist (i '(1 2 3 5 6 7))
     (if  (= i 3)
          (return 10))          ; Returns 10 when i = 3
     (print i))                 ; Prints 1 and then 2
```

The value of the last expression is returned by the block (unless modified by RETURN or RETURN-FROM). All other expressions in the block are thus only useful for their side effects.

7. Data Types

In the next two sections we will discuss conditionals and loops. To do so effectively, we need first briefly discuss data types and how Lisp handles boolean and logic.

As a reminder, it is important to note that in Lisp, data objects are typed, not variables. Any variable can have any Lisp object as its value.

Many Lisp objects belong to more than one type. The predicate TYPEP returns whether an object belongs to a given type, and the function TYPE-OF returns a type to which a given object belongs:

```
(typep "My String" 'string)    ; Returns True

(type-of "My String")          ; Returns (SIMPLE-ARRAY
                                 CHARACTER (9)). As we learn
                                 later, strings are an array
                                 of characters
```

Examples of Type Predicate Functions

ATOM	NULL	ZEROP	NUMBERP	EVENP
LISTP	ARRAYP	PLUSP	CHARACTERP	ODDP
SYMBOLP	PACKAGEP	MINUSP	STRINGP	ODDP

7.1 Boolean & Logic

The built-in types form a hierarchy of subtypes and supertypes. The symbol T (for truth) is the supertype of all types. We can express truth with any object other than NIL, the symbol for false:

```
(if 55 (print "True") (print "False"))        ; Prints True
```

The function AND returns NIL if any of its arguments are false and returns the value of the last argument if all arguments are true. The function OR returns the first argument that is true and NIL if no argument is true.

```
(and t (+ 1 2) (* 1 5))                       ; Returns 5
(or nil (+ 1 2) (* 1 5))                      ; Returns 3
```

7.2 Equality & Comparison

Common Lisp has a few different functions for testing equality of two objects. Most beginners should use EQUAL for non-numbers and = for numbers.

- EQ compares equality of memory addresses and is the fastest test. It is useful to compare symbols quickly and to test whether two cons cells are physically the same object. It should not be used to compare numbers.

- EQL is like EQ except that it can safely compare numbers for numerical equality and type equality. It is the default equality test in many Common Lisp functions.

- EQUAL is a general purpose test that, in addition to being able to safely compare numbers like EQL, can safely compare lists on an element by element basis. Lists are not unique and EQ and EQL will fail to return equality on equivalent lists if they are stored in different memory addresses.

- EQUALP is a more liberal version of EQUAL. It ignores case distinctions in strings, among other things.

- = is the most efficient way to compare numbers, and the only way to compare numbers of disparate types, such as 3 and 3.0. It only accepts numbers.

8. Conditionals

The five main conditionals in Common Lisp are IF, WHEN, UNLESS, COND and CASE. Conditionals with an implicit PROGN block allow for multiple forms within their bodies. To begin with an example of IF form (note there is no implicit PROGN):

```
(if (equal 5 (+ 1 4))
    (print "This is true")
    (print "This if false"))
```

Example of WHEN form (note there is an implicit PROGN):

```
(when (equal 5 (+ 1 4))
  (print "Print if statement is true")
  (print "Print this also"))
```

Example of UNLESS form (note there is an implicit PROGN):

```
(unless (equal 3 (+ 1 4))
  (print "Only print if condition is false")
  (print "Print this also"))
```

Example of COND form (multiple ifs, implicit PROGN). The form exits on the first true:

```
(cond ((equal 5 3) (print "This will not print"))
      ((equal 5 5) (print "This will print"))
      ((equal 5 5)
         (print "This will not print as the")
         (print "form exited at first true")))
```

Example of CASE form (implicit PROGN). Cases are literal and not evaluated:

```
(case (read)                    ; Try entering in 9 and then
  ((1 3 5 7 (* 3 3)) "Odd")     ; (* 3 3) at the read prompt
  (0                            ; Note implicit PROGN here
    (print "Zero")
    (print "Number"))
  (otherwise "Not a odd number < 10"))
```

9. Basic Loops

DOTIMES and DOLIST are basic loop macros for iteration. In the below example, DOLIST will iterate over the items of my-list and execute the loop body for each item of the list. my-variable holds the value of each successive item in the list during the iteration.

```
(dolist (my-variable my-list optional-result-form)
  body-form*)

(dolist (i '(1 2 3 5 6 7))
  (print i))
```

In the below example, DOTIMES will iterate my-variable from 0 to one less than the end-number supplied. If an optional-result-form is supplied, it will be evaluated at the end of the loop. Below is the structure of the macro, together with an example:

```
(dotimes (my-variable end-number optional-result-form)
  body-form*)

(dotimes (i 5 T)
  (print i))
```

10. Local Variables & Functions

LET and LET* are special operators that allow us to create local variables that can only be accessed within their closures. LET binds its variables in parallel such that you cannot refer to another variable in the LET form when setting the value of another. LET* binds its variables in sequentially, so that you can refer to the value of any previously bound variables. This is useful when you want to assign names to several intermediate steps in a long computation.

The LET form has the following syntax:

```
(let ((var-1 value-1)
      ...
      (var-n value-n))
  body-form*)
```

An example of LET* in use:

```
(let* ((x 5) (y (+ x x)))
  (print y))                                    ; Prints 10
```

As a general note on assignment, it is good programming style to avoid changing the value of local variables after they have been set. This makes our programs easier to follow as we do not need to track any changes in the values of our local variables. Thus, we should only use SETF on global variables and not alter variables created by LET and LET* forms within their closures.

10.1 Local Functions

Functions named by DEFUN are global functions that can be accessed anywhere. We can define local functions through FLET and LABELS, which are only accessible within their context. The scope of the functions in a FLET form is limited to the body of the form and function definitions are not visible to other functions defined by the FLET form. LABELS is equivalent to FLET except that the scope of the defined function names for LABELS encompasses the function definitions themselves as well as the body. Practically this means LABELS allows you to write recursive functions. The syntax of LABELS is:

```
(labels    ((fn-1 args-1 body-1)
            ...
            (fn-n args-n body-n))
  body-form*)
```

Functions defined within LABELS take a similar format to a DEFUN form. Within the body of the LABELS form, function names matching those defined by the LABELS refer to the locally defined functions rather than any global functions with the same names. Below is an example of a LABELS form that will return 12, the result of (+ 2 4 6), where 2, 4 and 6 are the results of evaluating the three local functions defined in the form.

```
(labels    ((first-function (x) (+ x x))
            (second-function (y) (* y y))
            (third-function (z) (first-function z)))
  (+ (first-function 1)
     (second-function 2)
     (third-function 3)))                       ; Returns 12
```

11. A Brief Detour - Code Formatting

Congratulations — you have now completed a basic introduction of Common Lisp! We will now take a brief detour to discuss conventions for formatting Lisp code, before returning to intermediate and advanced concepts in Lisp in the succeeding sections.

It is important to format Lisp code according to shared conventions. This will allow others to read your code and it will make your life easier as unformatted Lisp code is difficult to read.

11.1 Parentheses & Line Spacing

Horizontally space elements within Lisp forms like `(+ (+ 1 2) 3)`, with no whitespace around parentheses. Always put ending parentheses on the last line of code and not separately on newlines by themselves.

When a Lisp form does not fit on one line, consider inserting newlines between the arguments so that each one is on a separate line. However, do not insert newlines in a way that makes it hard to tell how many arguments the function takes or where an argument starts and ends.

11.2 Indentation

Indent your code the way a properly configured GNU Emacs does. In practice, this means relying on a Lisp editor (such as Emacs) that indents code automatically. Lisp editors have a complex set of indenting rules, but they can be generalised in most instances as follows:

- **Function arguments** are aligned with the first argument. If the first argument is on its own line, it is aligned with the function name
- **Bodies of forms** are indented two spaces
- **Distinguished ("special") arguments** are indented four spaces when on a newline

The purpose of indenting is to visually communicate nesting. This is most commonly achieved with the nesting of the bodies through the standard two spaces of indentation. Function arguments and distinguished arguments have a greater level of indentation to visually separate them and ensure they are not confused as part of a nested form.

11.3 Examples

Below are some examples that illustrate the above guidelines.

Good: Group ending parentheses on the last line of the form	**Bad**: Never place parentheses on their own lines

`(defun my-function (x)`
`  (if (< x 0)`
`  ("Negative")`
`  ("Non-positive")))`
	`(defun my-function (x)`
	`  (if (< x 0)`
	`  ("Negative")`
	`  ("Non-positive")`
	`  )`
	`)`

Good: Function arguments are aligned under the first argument	**Bad**: Without extra indentation, function calls can be confused for nested bodies

`(my-function first-arg`
`  second-arg`
`  third-arg)`
	`(my-function first-arg`
	`  second-arg`
	`  third-arg)`

Good: Bodies of forms should be nested two spaces	**Good**: Distinguished forms should be nested four spaces

`(when something`
`  (do-this)`
`  (and-this)`
`  (and-also-this))`
	`(with-slots (a b)`
	`  **(distinguished-form)**`
	`  (print a)`
	`  (print b))`

Note in the first example above, the IF form is indented in a similar manner to a function call. This is purely coincidental: the test, then and else forms are distinguished arguments and indented four spaces. This coincidentally happens to line them up under the first argument.

To doubly emphasise the point of function calls and distinguished arguments having special indentation, note that an IF form would be very confusing with standard indentation as we may incorrectly assume the last two forms are to be evaluated consecutively as part of the then form:

```
(if (< x 0)
  (+ 1 5)
  (+ 4 6))
```

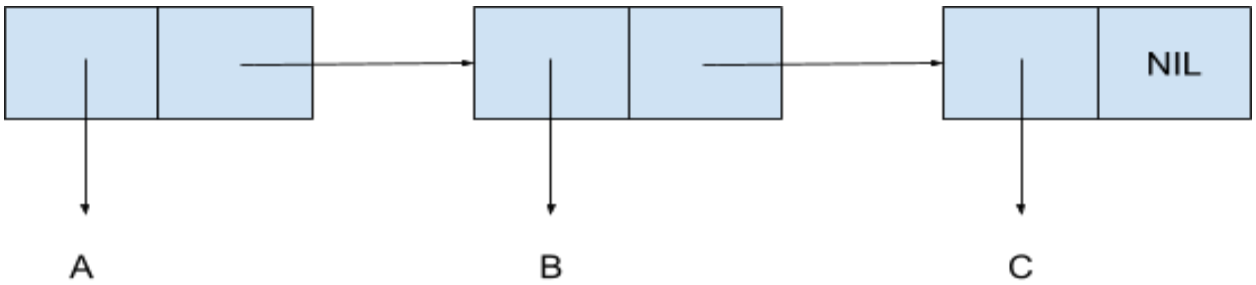
12. Lists & List Manipulation

Lisp stands for **Lis**t **P**rocessor and a deeper understanding of lists is essential to proficiency in the language. Our earlier definition of atoms & lists focused on their printed representation. We begin this section with detail on their internal representation.

Inside computer memory, lists are organized as chains of cons cells. CONS are a pair of pointers, the first cell is called the CAR and the second the CDR. The CAR points to some data structure (e.g. an integer, string, or even another list), while the CDR points to either another CONS or to the empty list NIL.

Lists are thus defined recursively to be either empty lists or CONS whose CDR components are lists. The list (A B C) is comprised of the following (observe the recursive definition in action):

Recursive definition of the list (A B C): A cons with car of A and cdr of (B C) → A cons with car of B and cdr of (C) → A cons with car of C and cdr of NIL → The empty list NIL



Atoms are defined simply as not cons. The empty list NIL is both an atom and a list. We can access the first and second con cells of a list with the CAR and CDR accessors:

```
(car '(1 2 3))                    ; Returns 1
(cdr '(1 2 3))                    ; Returns (2 3)
```

We can join atoms or lists into a pair of cons cells with the CONS function:

```
(cons 1 '(2 3))                   ; Returns (1 2 3)
(cons '(1 4) '(2 3))              ; Returns ((1 4) 2 3)
```

Note in our last example, the CAR of the generated list is yet another list (1 4) as we passed a list as the CAR argument of the CONS function. This gives a glimpse into creating nested lists.

When printing a list in parenthesis notation, Lisp starts by printing a left parenthesis followed by all the elements, separated by spaces. If the list ends in NIL, Lisp prints a right parenthesis. A proper list is a cons cell chain ending in NIL.

If the list does not end in NIL, before printing the right parenthesis Lisp prints a space, a period, another space, and the atom that ends the chain. A list not ending in NIL is called a dotted list:

```
(cons 1 2)                                  ; Returns (1 . 2)
```

12.1 Building & Copying Lists

The LIST function allows us to create lists of more than two elements by consing onto NIL. The COPY-LIST function takes a list as its argument and returns a copy of it. The APPEND function returns a concatenation of the elements of any number of lists supplied as its argument.

```
(list 3 'a 'b 'c 'd)                        ; (3 A B C D)
(list '(a b c d) 3)                         ; ((A B C D) 3)
(append '(a b c d) '(a b c d))              ; (A B C D A B C D)
(append '(a b c d) 3)                       ; (A B C D . 3)
```

12.2 Accessing List Elements

The NTH function can access an element at a given position in a list while the function NTHCDR is used to get CDR at a given position. In addition, we can FIRST - TENTH to get the 1st to 10th elements of a list, while LAST will give the last CDR in a cons cell.

```
(first '(a b c d e f g))                    ; Returns A
(fifth '(a b c d e f g))                    ; Returns E
(tenth '(1 2 3 4 5 6 7 8 9 11))             ; Return 11
(last '(a b c d e f g))                     ; Returns (G)
                                             (the last CDR)
```

Finally, lists are sequences and there are a number of useful functions for sequences that can be used on lists (these are discussed later in this guide).

12.3 Lists as Sets

Lists are a good way to represent small sets. The function MEMBER checks if an element is part of a list and returns the part of the list beginning with the element if it is found.

```
(member 'b '(a b c))                    ; Returns (B C)
```

Recall that lists are not unique and lists with the same elements can be stored in different parts of computer memory. MEMBER utilises EQL for its comparison, and will not return a match where the two lists are stored in different parts of memory. To achieve this, i.e. to compare lists on an element by element basis, utilise the following configuration of the MEMBER function:

```
(member 'b '(a b c) :test #'equal)      ; Element-wise test
```

We can also specify a function to be applied to each element before the comparison. In the below example, we test if there is an element whose CAR is the symbol B:

```
(member 'b '((a) (b) (c d)) :key #'car) ; ((B) (C D))
```

MEMBER-IF allows us to search for an element satisfying an arbitrary predicate. For example, in the below, we search for odd numbers in the list and return the part beginning with the first odd:

```
(member-if  #'oddp '(2 3 4))            ; (3 4)
```

ADJOIN joins an object onto a list only if it is not already a member:

```
(adjoin 'b '(a b c))                    ; Returns (A B C)
(adjoin 'z '(a b c))                    ; Returns (Z A B C)
```

The below examples illustrate set union, intersection and complement operations on exactly two lists. SET-DIFFERENCE returns a list of elements of the first list that do not appear in the second list. Also note that since there is no notion of ordering in a set, the below functions do not necessarily bother to preserve the order of elements found in the original list.

```
(union '(a b c) '(c b s))               ; Returns (A B C S)
(intersection '(a b c) '(c b s))        ; Returns (C B)
(set-difference '(a b c) '(c b s))      ; Returns (A)
```

12.4 Association Lists

We can use a list of conses (remember a cons consists of two elements, its car and its cdr) to represent mappings. ASSOC is used to retrieve the value associated with a particular key.

Below is an example of defining and retrieving from an assoc-list (which is just a list of conses):

```
(defparameter  mapping-table
               '((+ . "add") (- . "subtract")))

(assoc '+ mapping-table)            ; Returns (+ . "add")
```

12.5 Pushdown Stacks

We can use lists as **pushdown** stacks. The macro PUSH can be used to push an element to the front of the list, while the macro POP can remove and return the first element of the list. Both are **destructive** operations as they directly change the original lists in question. For example:

```
(defparameter my-list '(2 3 4))

(push 1 my-list)          ; Returns (1 2 3 4)
my-list                   ; Returns (1 2 3 4)

(pop my-list)             ; Returns 1, the car of the list
my-list                   ; Returns (2 3 4)
```

13. More on Functions

Functions in Lisp are first-class objects that generally support all operations available to other data objects, such as being modified, passed as an argument, returned from a function and being assigned to a variable.

The FUNCTION special operator (shorthand #') returns the function object associated with the name of function that is supplied as an argument:

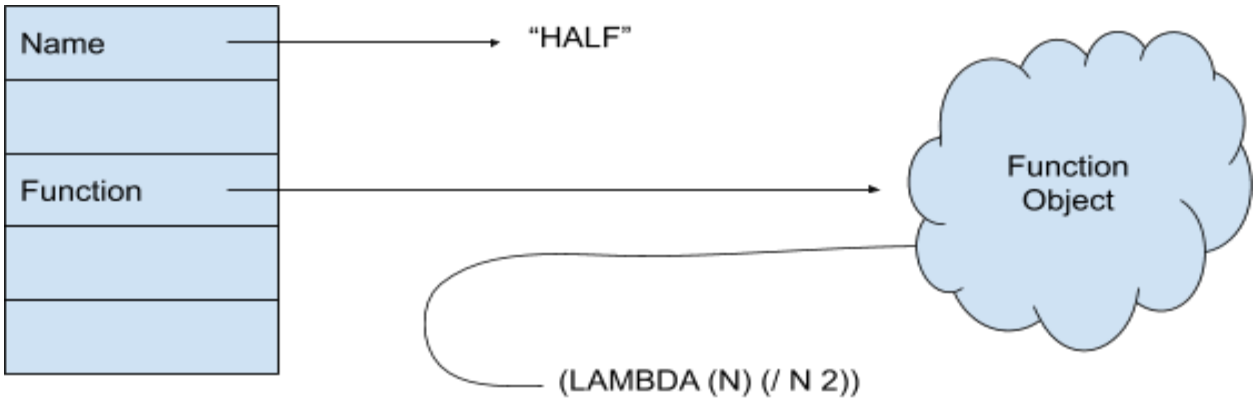
```
(function +)         ; Returns the function object

#'+                  ; Equivalent syntax
```

When you type the above in your toplevel, your Lisp implementation will print the external representation of the function object (implementations are free to choose whatever external representation they like), such as #<FUNCTION +>. Internally however, a built-in function like + is likely to be a segment of machine language. CONS or + are examples of symbols used to name built-in Lisp functions. The symbol CONS has a pointer in its function cell to a "compiled code object" that represents the machine language instructions for creating new cons cells.

For user defined functions, the DEFUN macro is used to name a function with a symbol. For example, in the below example, the symbol HALF names the function, whilst the symbol itself is named by the string "HALF". The function cell of the symbol points to a function object that is the real function. Exactly what this function object looks like depends on which implementation of Common Lisp you're using, but as the diagram indicates, there's probably a lambda expression in there somewhere.

Example showing how symbols point to function objects



APPLY takes a function and a list of arguments for it and returns the result of applying the function to its arguments. Note how we have to use to sharp-quote (#') to pass the + function as an object into the APPLY function. Without doing so, Lisp will return an error as it will try to evaluate +, which is not legally permissible in the below example.

```
(apply #'+ '(1 2 3))                    ; Returns 6
```

The function FUNCALL is similar to APPLY, but allows us to pass arguments individually and not packaged as a list:

```
(funcall #'+ 1 2 3)                     ; Returns 6
```

We can use APPLY and FUNCALL to evaluate lambda expressions, as lambda expressions are nothing but lists which can be used directly in place of function names.

Below is an example of passing a lambda expression to FUNCALL:

```
(funcall #'(lambda (x y z) (+ x y z)) 1 2 3) ; Returns 6
```

As a closing remark, Lisp programs are nothing but lists themselves. The function EVAL takes an expression, evaluates it and returns its value. In fact, our toplevel is nothing but a read-eval-print loop (hence known as REPL). Calling EVAL ourselves is not advisable as it is inefficient (lists are evaluated at run-time and not as compiled coded) and as it does not handle lexical context (EVAL within a LET cannot refer to variables established by the LET). Indeed, one of the only places where it is appropriate to use EVAL is in a top-level loop.

```
(eval '(+ 1 2 3))                                ; Returns 6
```

13.1 Function Parameters

By default, a function call must supply values for all parameters that feature in the function definition. We can modify this behaviour with the &optional, &key and &rest tokens. The &optional token allows to distinguish between required parameters, placed before the &optional token, and optional parameters, placed after the token:

```
(defun make-a-list (a b c d &optional e f g)
  (list a b c d e f g))

(make-a-list 1 2 3 4 5)        ; Returns (1 2 3 4 5 NIL NIL)
```

One drawback of the &optional token, using the above as an example, is that we need to supply values for E and F if we want to supply the value for G, as arguments in a function call are assigned to the parameters in order. To overcome this, we utilise the &key token to be able to specify which optional parameter we want to assign a value to. Below is an example of this,

```
(defun make-a-list-2 (a b c d &key (e 1) f g)
  (list a b c d e f g))

(make-a-list-2 1 2 3 4 :g 7)   ; Returns (1 2 3 4 1 NIL 7)
```

The preceding example also shows how we can supply a default value to an optional (setting E to 1 if no value for E is provided). When we called this function in the above, we set G to 7 and E also defaulted to 1. As no value was supplied for F, it defaulted to NIL.

In general, &key is preferable to &optional as it allows us to have greater control in our function calls. It also makes code easier to maintain and evolve as we can add new parameters to a function without affecting existing function calls (useful when writing libraries that are already being used by other programs).

Finally, the &rest token, placed before the last variable in a parameter list, allows us to write functions that can accept an unknown number of arguments. The last variable will be set to a list of all the remaining arguments supplied by the function call:

```
(defun make-a-list-3 (a b c d &rest e)
  (list a b c d e))

(make-a-list-3 1 2 3 4 5 6 7 8)     ; (1 2 3 4 (5 6 7 8))
```

We can utilise multiple tokens in the same function call, as long as we declare them in order. First the names of required parameters are declared, then the optional parameters, then the rest parameter and finally the keyword parameters are declared.

13.2 Multiple-Value-Binds

Until now, we have only considered functions that return one value. In certain circumstances it is useful for a function to return several values, without having to build an overarching structure to contain them all. This is achieved through multiple-value-binds.

The VALUES function returns multiple values and can be used as the last expression in the body of a function. The below example returns 1, NIL and 6 (individually, not as a list):

```
(values 1 nil (+ 2 4))
```

If a VALUES function is supplied as an argument to a form which is only expecting one value, the first value returned by the VALUES function is used and the rest are discarded:

```
(+ 5 (values 1 nil (+ 2 4)))                    ; Returns 6
```

The MULTIPLE-VALUE-BIND macro is used to **receive** multiple values. The first argument of this macro is the variables and the second is the expression that returns their values. We can then use these values in the body of the multiple-value-bind macro. Below is an example.

```
(multiple-value-bind (x y z) (values 1 2 3)
  (list x y z))                          ; Returns (1 2 3)
```

If there are more variables than values, the leftover variables will be bound to NIL. If there are more values than variables, the extra values will be discarded. Finally, you can pass multiple values as arguments to a function using the MULTIPLE-VALUE-CALL special operator:

```
(multiple-value-call #'+ (values 1 2 3))      ; Returns 6
```

13.3 Applying Functions To Elements of a List

MAPCAR takes a function and one or more lists and returns a list of the results of applying the function to elements taken from each list. A function with multiple arguments takes one element from each list, as in the second and third examples below.

```
(mapcar   #'(lambda (x) (+ x 1))
          '(1 2 3))                        ; Returns (2 3 4)

(mapcar   #'(lambda (x y) (+ x y))
          '(1 2 3) '(5 10 15))             ; Returns (6 12 18)

(mapcar   #'+
          '(1 2 3) '(5 10 15) '(10 20 30))    ; (16 32 48)
```

A similar function is MAPLIST, which works on successive cdrs of the list:

```
(maplist  #'(lambda (x) x)
          '(1 2 3))              ; Returns ((1 2 3) (2 3) (3))
```

14. Arrays & Sequences

The function MAKE-ARRAY allows us to create arrays. For example, we can create a 2 x 3 array as follows:

```
(defparameter my-array (make-array
                        '(2 3)
                        :initial-element nil))
```

The functions AREF and SETF allow us to access elements and set them with values:

```
(aref my-array 0 0)                     ; Returns NIL
(setf (aref my-array 0 0) 'b)           ; Set (0,0) to B
(aref my-array 0 0)                     ; Returns B
```

The functions ARRAY-RANK and ARRAY-DIMENSION retrieve the the number of dimensions and the number of elements in a given dimension respectively:

```
(setf my-array
        (make-array '(2 3)
                    :initial-element '((1 2 3) (1 2 3))))

(array-rank my-array)                   ; Returns 2
(array-dimension my-array 0)            ; Returns 2
(array-dimension my-array 1)            ; Returns 3
```

We use :INITIAL-ELEMENT to set the value of every element of an array to the provided argument, while we use :INITIAL-CONTENTS to set the array to the object provided. A one-dimensional array is a vector and can be created with either of the following.

```
(vector "a" 'b 3)

(defparameter my-vector
               (make-array 3 :initial-contents '("a" 'b 3)))
```

Finally, we can create a literal array using the #na syntax, where n is the number of dimensions:

```
#2a((b nil nil) (1 2 3))  ; Returns ((B NIL NIL) (1 2 3))
```

Strings are vectors of characters, denoted with double quotes (e.g. "my-string"). Strings evaluate to themselves. A **character** such as c is denoted as #\c. Each character has an associated integer that is usually (but not necessarily) its ASCII number:

```
(char-code #\c)                    ; Returns 99
(code-char 99)                     ; Returns #\c
```

14.1 Sequences

The type **sequence** includes both **lists** and **vectors** (and therefore **strings**). Sequences have many useful functions:

```
(length '(a b c d e f))            ; Returns 6
(reverse '(a b c d e f))           ; Returns (F E D C B A)

;; Returns (C R T) (a new original list unaffected):
(remove 'a '(c a r a t))

;; Returns "cbdra" (preserves only the last of each):
(remove-duplicates "abracadabra")
```

We use SUBSEQ to get a portion of a list. Its arguments are a list, the starting position and an optional ending position (which is not to be included in the subsequence):

```
(subseq '(a b c d e f) 1 4)        ; Returns (B C D)
```

SORT takes a sequence and a comparison function of two arguments and **destructively** (i.e. by modifying the original sequence) returns a sequence sorted according to the function:

```
(sort '(1 4 2 5 6) #'>)            ; Returns (6 5 4 2 1)
```

The functions EVERY and SOME test whether a sequence satisfies a provided predicate:

```
(every #'oddp '( 1 2 5))           ; Returns NIL

(some #'oddp '( 1 2 5))            ; Returns T

(every #'> '(1 3 5) '(0 2 4))      ; Returns T
```

14.2 Keyword Arguments

Many sequence functions take one or more keyword arguments from the below table. For example, we can use POSITION to return the position of an element within a sequence (or NIL if not found) and use keyword arguments to determine where to begin the search:

```
(position #\a "fantasia" :start 3 :end 7)     ; Returns 4
```

Parameter	Purpose	Default
:key	A function to apply to each element	identity
:test	The test function for comparison	eql
:from-end	If true, work backwards	nil
:start	Position at which to start	0
:end	Position, if any, at which to stop	nil

The function REDUCE is useful to extend functions that only take two variables. It takes two arguments, a function (which must take exactly two values) and a sequence. The function is initially called on the first two elements of the sequence, and thereafter with each successive element as the second argument. The value returned by the last call is the value returned by the REDUCE function. For example, the below returns (A), the intersection of these three lists:

```
(reduce #'intersection '((b r a d) (b a d) (c a t)))
```

15. Hash Tables

A **hash table** is a way of associating pairs of objects, like a dictionary. The objects stored in a hash table or used as keys can be of any type. We can make hastables with MAKE-HASH-TABLE and retrieve values associated with a given key with GETHASH:

```
(defparameter my-hash-table (make-hash-table))

(gethash 'color my-hash-table) ; Returns NIL as not yet set
```

Similar to other structures, we use SETF to set values. Hash tables can accommodate any number of elements, because they are expanded when they run out of space. We can remove values with REMHASH.

```
(setf (gethash 'color my-hash-table) 'red)    ; Returns RED

(remhash 'color my-hash-table)
```

Finally, the function MAPHASH allows you to iterate over all entries in the hash table. Its first argument must be a function which accepts two arguments, the key and the value of each entry. Note that due to the nature of hash tables you can't control the order in which the entries are provided to MAPHASH (or other traversing constructs):

```
(maphash #'(lambda  (key value)
                    (format t "~A = ~A~%" key value))
                   my-hash-table)
```

16. Structures

Common Lisp provides the DEFSTRUCT facility for creating named data structures with named components. This makes it easier to manipulate custom data objects as we can refer to their components by name. Constructor, access and assignment constructs are automatically defined when a data type is defined through DEFSTRUCT.

Consider the below example of defining a data type for rectangles. DEFSTRUCT defines RECTANGLE to be a structure with two fields, height and width. The symbol RECTANGLE becomes the name of a data type and each rectangle will be of type RECTANGLE, then STRUCTURE, then ATOM and then T. DEFSTRUCT generates four associated functions:

1. RECTANGLE-HEIGHT and RECTANGLE-WIDTH to access elements of the structure
2. RECTANGLE-P to test whether an object is of type rectangle
3. MAKE-RECTANGLE to create rectangles
4. COPY-RECTANGLE to create copies of rectangles

Example of a Rectangle Structure

```
(defstruct rectangle
  (height)                    ; Height will default to NIL
  (width 5))                  ; Width will default to 5

(defvar rectangle-1)

(setf rectangle-1 (make-rectangle :height 10 :width 15))

(rectangle-height rectangle-1)                ; Returns 10

(setf (rectangle-width rectangle-1) 20)       ; Returns 20

(defvar rectangle-2)

(setf rectangle-2 (make-rectangle))

rectangle-2    ; Prints #S(RECTANGLE :HEIGHT NIL :WIDTH 5)
```

There are some advanced initialization options that we will not discuss here. Note that the the #S syntax can be used to read instances of rectangle structures.

17. Reading & Writing to Files

The WITH-OPEN-FILE macro is used to read & write to files and then close the file. Streams are Lisp objects representing sources and/or destinations of characters. To read from or write to a file, you open it as a stream. By default, input is read from the stream *standard-input* and output is recorded in *standard-output*. Initially they will be the same place - the toplevel.

Below is an example opening a file as my-stream and then reading from it. The NIL in the below inhibits end of file errors.

```
(with-open-file (my-stream "/Users/ashokkhanna/test.txt")
  (format t "~a~%" (read-line my-stream nil)))
```

Below is an example opening a file as my-stream and then writing to it.

```
(with-open-file (my-stream "/Users/ashokkhanna/test.txt"
                           :direction :output
                           :if-exists :append)
  (format my-stream "~a~%" "Hello, World!"))
```

The following open arguments can be supplied to the WITH-OPEN-FILE macro:

- Write to a file instead of reading `:direction :output`
- Create a file if it does not exist `:if-does-not-exist :create`
- Replace file that exists `:if-exists :supersede`
- Overwrite file `:if-exists :overwrite`
- Write to end of file `:if-exists :append`

18. Advanced Looping

The DO macro is a very powerful and flexible iterator. It looks like this:

```
(do ((var1 init1 step1)
     ...
     (varn initn stepn))
    (end-test result-forms*)
  body-forms*)
```

Below is an example of the DO loop, together with a detailed step through. This example will return 81 and print 1, 0, 1, 4, 9, 16, 25, 36, 49 and 64 on newlines. During each iteration, loop-step is increased by one while square is set to the square of loop-step.

```
(do ((loop-step 0 (+ loop-step 1))
     (square 1 (* loop-step loop-step)))
    ((= 10 loop-step) square)                    ; Stop at 10
  (print square))                ; Print square at each step
```

Step Through

1. The init forms are evaluated at the beginning of the loop and bound to the variables. In the above example, loop-step is bound to 0 while square is bound to 1.

2. The end-test form is evaluated at the beginning of each iteration. If it evaluates to NIL, the iteration proceeds and the body of the loop is executed. Hence, in the above example you see 1 printed on the first iteration, as that is the starting value of the square variable.

3. After all the body forms have been evaluated, and before each subsequent iteration of the loop, the step forms will be evaluated and their values will be bound to the variables.

   In a DO loop, the step forms can refer to other variables defined by the loop, but the value they receive for these variables is the value **before** the step forms are evaluated. Thus, in the first update, square is set to 0 as that is the value of loop-step before the step forms are evaluated.

4. This can be seen in the printed output of 0 provided by the second iteration of the loop.

5. Note in a DO loop, the init forms cannot refer to other variables defined in the loop as they have not yet been bound to a value (the DO loop, similar to LET, binds values in parallel).

6. After 10 iterations, the loop-step variable will have a value of 10. Accordingly, the end-test form will return T. When this occurs, the result-forms are evaluated and the value of the last result form is returned as the value of the loop.

7. In our example above, the value of square will be 81 at this point, being the square of the last value of loop-step (9). This is the value returned by the loop.

8. Note that the body-forms are not evaluated when the end-test is true, and the last printed output is 64 in our example.

The DO* macro is similar to a LET* form and binds its variables in sequence. Therefore a variable can access the latest value of a previously defined variable in either the initial or step forms.

Below is an near-identical example of the above, but with a DO* loop. The DO* loop will return 100 and print 0, 1, 4, 9, 16, 25, 36, 48, 64 and 81 on newlines.

```
(do* ((loop-step 0 (+ loop-step 1))
      (square loop-step (* loop-step loop-step)))
     ((= 10 loop-step) square)
  (print square))
```

As a final example, consider the below, where we switch the lexical positions of loop-step and square within the loop. This example will return 81 and print 0, 0, 1, 4, 9, 16, 25, 36, 49, 64 and 81. Observe how square is now accessing the prior value of loop-step, as it is evaluated before loop-step and does not have access to its current value. This is a reminder that the DO* loop performs its bindings in sequence.

```
(do* ((square 0 (* loop-step loop-step))
      (loop-step 0 (+ loop-step 1)))
     ((= 10 loop-step) square)
  (print square))
```

19. Scope & Extent

We will briefly introduce and discuss the concepts of scope and extent. Before we begin, as a general rule, local variables and functions (defined by LET, LET*, LABELS, FLET) can only be accessed within their closures, whilst global names (defined by DEFVAR, DEFPARAMETER, DEFCONSTANT, DEFUN) are accessible everywhere.

It is best practice to avoid using the same names for local and global variables and functions.

Scope refers to the textual region of the program within which references may occur, whilst extent refers to the interval of time during which references may occur. For example, the scope of the parameter X in the below is the body of the DEFUN form and its extent is the interval from the time the function is invoked to the time it is exited:

```
(defun copy-cell (x)
  (cons (car x) (cdr x)))
```

Accordingly, there are four permutations of scope and extent as listed below.

1. **Lexical scope**: Here references to the established entity can occur only within certain program portions that are lexically (that is, textually) contained within the establishing construct. Typically the construct will have a part designated the body, and the scope of all entities established will be (or include) the body.

2. **Indefinite scope**: References may occur anywhere, in any program.

3. **Dynamic extent:** References may occur at any time in the interval between establishment of the entity and the explicit disestablishment of the entity. As a rule, the entity is disestablished when execution of the establishing construct completes or is otherwise terminated.

4. **Indefinite extent**: The entity continues to exist as long as the possibility of reference remains. (An implementation is free to destroy the entity if it can prove that reference to it is no longer possible. Garbage collection strategies implicitly employ such proofs.)

Variable bindings and bindings of local function names have lexical scope and indefinite extent, whilst bindings declared to be special (such as DEFVAR and DEFPARAMETER) have "**dynamic scope**" (indefinite scope and dynamic extent). Named constants such as NIL and PI have indefinite scope and indefinite extent.

The binding rule for dynamic scope in Lisp is as follows: a use of a name is bound to the most recent declaration of that name that is still live (i.e. we first look for a local definition of a variable, if it isn't found, we look up the calling stack for a definition).

20. Packages

In large Lisp systems, with modules written by many different programmers, accidental name collisions become a serious problem. Packages are used to overcome this issue.

A package is a data structure that establishes a mapping from print names (strings) to symbols. The string-to-symbol mappings available in a given package are divided into two classes,

external and internal. Within a given package, a name refers to one symbol or to none; if it does refer to a symbol, then it is either external or internal in that package, but not both.

- **External symbols are part of the package's public interface to other packages**. and are to be chosen with care. They are advertised to users of the package.

- **Internal symbols are for internal use only**, and are normally hidden from other packages. Most symbols are created as internal symbols; they become external only if they appear explicitly in an export command for the package.

At any given time, only one package is current. This package is used by the Lisp reader in translating strings into symbols. The current package is, by definition, the one that is the value of the global variable *package*.

It is possible to refer to symbols in packages other than the current package through the use of package qualifiers in the printed representation of the symbol. For example, FOO:BAR, when seen by the reader, refers to the symbol whose name is BAR in the package whose name is FOO. This is technically only true if BAR is an external symbol of FOO. A reference to an internal symbol requires the intentionally clumsier syntax FOO::BAR.

Symbols in the keyword package have two unique properties: (1) they always evaluate to themselves and (2) they can be referred anywhere simply as :X instead of KEYWORD:X.

21. Symbols & Variables

A deeper understanding of symbols and variables is useful as you progress to more advanced Lisp programs and we will aim to cover some of their detail here.

21.1 Internal Representation of Symbols

Symbols are Lisp data objects that serve several purposes and have several interesting characteristics. Conceptually, a symbol is a block of five pointers. Internally, symbols are composed of five cells: the name, value, function, plist, and package cells. A symbol may have uppercase letters, lowercase letters, numbers, certain special characters or a mixture in its print name; it however cannot be a number.

Symbols are unique, meaning there can be only one symbol in the computer's memory with a given name. Every object in the memory has a numbered location, called its address. Since a symbol exists in only one place in memory, symbols have unique addresses.

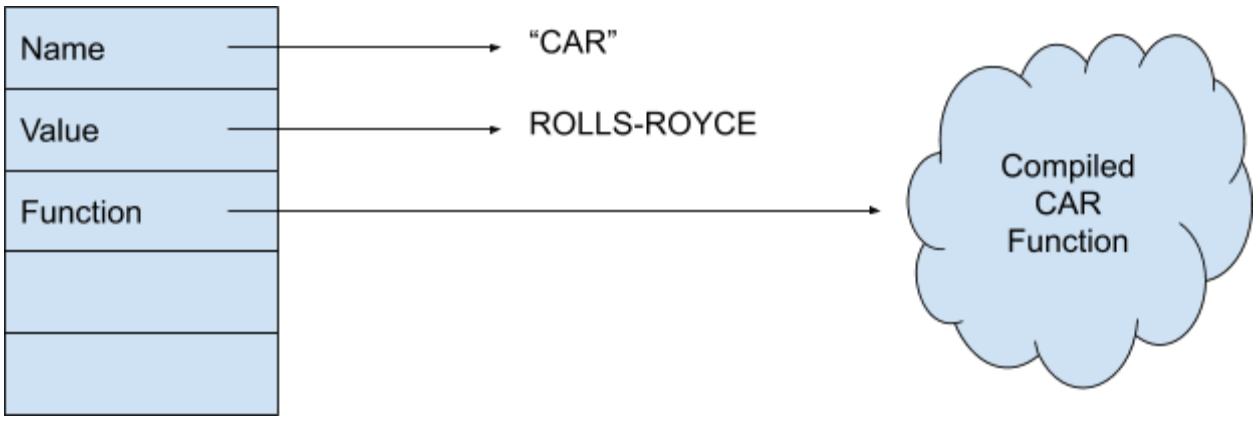
Every object of type symbol has a name, called its print name: Given a symbol, one can obtain its name in the form of a string. Conversely, given the name of a symbol as a string, one can obtain the symbol itself. (More precisely, symbols are organized into packages, and all the symbols in a package are uniquely identified by name).

21.2 Variables & Functions can share a symbol name

Because symbols have separate function and value cells, we can have a variable and a function with the same name. Common Lisp determines whether a symbol refers to a function or a variable based on the context in which it appears. If a symbol appears as the first element of a list that is to be evaluated, it is treated as a function name. In other contexts it is treated as a variable name.

In the below example, the symbol CAR is associated with both the CAR function and the value "ROLLS-ROYCE". Thus, the form (CAR '(A B C)) will call the CAR function, which returns A. On the other hand, the form (LIST 'A 'NEW CAR) references the global variable CAR and produces the result (A NEW "ROLLS-ROYCE").

Example of a Symbol with both Variable & Function assigned — Credits David David S. Touretzky, Common Lisp: A Gentle Introduction to Symbolic Computation



21.3 What are Variables?

As noted at the start of this guide, variables are a place where a value is stored. Variables represent references to objects and assigning a variable a new value changes what object the variable refers to but has no effect on the previously referenced object. However, if a variable holds a reference to a mutable object, you can use that reference to modify the object, and the modification will be visible to any code that has a reference to the same object.

Each time a function is called, Lisp creates new bindings to hold the arguments passed by the function's caller. A binding is the runtime manifestation of a variable. A single variable - the thing you can point to in the program's source code - can have many different bindings during a run of the program. A single variable can even have multiple bindings at the same time; parameters to a recursive function, for example, are rebound for each call to the function.

Much of Lisp's terminology for variables is a holdover from the days when dynamic scoping was the norm. For historical reasons some writers talk about ''binding a variable'' when they mean ''creating a new variable.'' But people also say ''unbound variable'' when they mean ''unassigned variable.'' Binding does not refer strictly to assignment; that is one of the major sources of terminological confusion in Lisp. Non-global lexical variables always have values, but it is possible for global or special variables to exist without a value. We won't get into the arcane details of that here.

21.4 Interplay of Symbols & Variables

Variables are named by symbols but they are not symbols themselves. Symbols are related to variables in two very different ways:

- Special variables (i.e. global variables), such as those established by DEFVAR and DEFPARAMETER, are stored with a symbol with the same name. The value of the special variable is stored within the value cell of the symbol

- Lexical variables and lexical function definitions are also named by symbols, but here the symbol is only an object identifier, and not a "full symbol". In this role, only the symbol's name is significant and Common Lisp provides no operations on symbols that can have any effect on a lexical variable or a lexical function definition.

Special Variables DEFVAR & DEFPARAMETER	**Lexical Variables & Functions** LET, LET*, FLET, LABELS
Variable are named by the symbol, whose value cell holds the variables value. A symbol evaluates to the value of the variable it holds. Any part of the program can access the variable by referencing the associated symbol that exists within the global Lisp user package. Hence, the variable is a global variable.	Variables are named by the symbol, but there is no local symbol that is holding the variable's value. By compile time, this reference is translated to location in memory and there will be no trace of the symbol. Parts of the program outside of the lexical scope cannot reference the variable through its symbol name as no symbol is created within the global Lisp user package.

21.5 Symbols & Property Lists (PLIST)

Symbols have a component called the property list, or plist. By convention this is always a list whose even-numbered components (calling the first component zero) are symbols, here functioning as property names, and whose odd-numbered components are associated property values (either a value or a function, but not both). Functions are provided for manipulating this property list; in effect, these allow a symbol to be treated as an extensible record structure.

The function GET is used to retrieve a value associated with a key in a symbol's property list and we can use it in conjunction with SETF to set values:

```
(get 'symbol-name 'my-key)     ; Uses eql to compare keys

(setf (get 'symbol-name 'my-key) 3)      ; Set my-key to 3
```

To illustrate these concepts, the below example associates a function with a plist key and then we retrieve and apply the function:

```
(setf (get 'symbol-name 'my-key) (lambda (x) (+ x 100)))

(funcall (get 'symbol-name 'my-key) 1)        ; Returns 101
```

22. Lambda Expressions

As noted earlier, there are four types of compound forms evaluated by Lisp: special forms, macro forms, function forms and lambda forms. A lambda form is similar to a function form, except the function name is replaced by a lambda expression. As an example, the below two are equivalent. Lambda expressions can be used to utilise unnamed functions.

Function Form	**Lambda Form**

`(defun my-function (x)`
`  (+ x 100))`
 	
`(my-function 1)  ; Returns 101`
	`((lambda (x)`
	`  (+ x 100))`
	`1)  ; Returns 101`

More specifically, a lambda expression is a list which can be used in place of a function name in certain contexts to denote a function by directly describing its behavior rather than indirectly by referring to the name of an established function. Its name derives from the fact that its first element is the symbol LAMBDA. The second element of a lambda expression must be an argument list and its remaining elements constitute the body of the function.

In a slightly confusing manner, LAMBDA is also a Common Lisp macro. Depending on context, LAMBDA can refer either to the symbol LAMBDA or the macro LAMBDA. LAMBDA will be evaluated as a macro when it is the first element of a macro form. LAMBDA will be treated as a symbol when it is used as part of a lambda expression such as in the lambda form in the above table or as an argument to FUNCTION below.

Below is an example of LAMBDA evaluated as a macro:

```
(funcall (lambda (x) (+ x 100)) 1)          ; Macro call
```

The LAMBDA macro expands its arguments to (function (lambda ...)). The above expression is expanded to:

```
(funcall (function (lambda (x) (+ x 100))) 1) ; Returns 101
```

We can supply lambda expressions as arguments to FUNCTION. Thus, in the above, LAMBDA refers to the symbol LAMBDA. The FUNCALL function above will evaluate the lambda form and returns 101.

23. Macros

Macros are special kinds of functions whose arguments are not evaluated. Macro functions must return Lisp expressions, which are then evaluated. Macros are important in the writing of good code: they make it possible to write code that is clear and elegant at the user level but that is converted to a more complex or more efficient internal form for execution. Reddit user ramenbytes assisted with this section.

23.1 Basic Syntax

Macros are defined by DEFMACRO, which is very similar to DEFUN:

```
(defmacro macro-name (parameters*)
  "Optional documentation string."
  body-forms*)
```

Below is an example of a macro definition and a macro call. This macro will convert the form (MY-ADD-MACRO X) to (+ 1 2 3 X) and then the subsequent macro call will expand the function, inserting 8 into X (during READ TIME - refer section below), and evaluate (+ 1 2 3 8) to return 14.

```
(defmacro my-add-macro (x)
  `(+ 1 2 3 ,x))

(my-add-macro 8)                        ; Returns 14
```

The function MACROEXPAND-1 takes a macro call and generates its expansion:

```
(macroexpand-1 '(my-add-macro x))       ; Returns (+ 1 2 3 X)
```

23.2 Use of Backquote

Backquotes `` ` `` prevent evaluation in a similar manner to regular quotes `'`, but have additional functionality with the use of the unquote operators `,` (comma) and `,@` (comma-at) that provide the ability to turn evaluation back on. As an example:

```
(defparameter x 2)
'(+ 1 2 3 x)                    ; Returns (+ 1 2 3 X)
`(+ 1 2 3 ,x)                   ; Returns (+ 1 2 3 2)
```

23.3 Compile Time, Read Time, Run Time

When a function is being compiled, any macros it contains are expanded at compilation time. Therefore macro definitions must be seen by the compiler before their first use. To understand how macros work, we need to understand the three stages of evaluation - compile time, read time and run time.

We will carefully walk through the examples on the following two pages to illustrate these concepts. For this, we have created three simple macros, the bad macro, the good macro and the best macro and applied them to three examples.

1. **Macros are expanded at compilation time**. During this phase, they take any arguments supplied in the macro definition and **insert them without evaluation** into the form (as per the macro definition). For example, in Example 2 below, the bad macro returns an error "X is not a number" during the macroexpansion phase as it has received an unevaluated symbol X and is trying to add that to a series of numbers.

2. That said, we can **control evaluation through reader macros** like quote `'`, backquote `` ` `` and comma `,`. These occur at READ TIME, i.e. when they are read by the lisp reader and parsed into lisp objects. For both the good macro and the best macro in the below examples, we use quotes and backquotes to prevent evaluation during read time and the lisp reader supplies the unevaluated objects for the macroexpansion.

   Note that within the best macro, we use comma to turn evaluation back on, and thus, during READ TIME, we evaluate the argument X of the macro. Without this forced evaluation, the argument would be inserted into the macro expansion without evaluation, as per point 1 above. Hence, in example 1 below, you can see the macro expansion of the best macro is (+ 1 2 3 8) vs. the macro expansion of the good macro which is (+ 1 2 3 X), the latter utilising the unevaluated symbol X whilst the former evaluating X to 8 and supplying this to the macro expansion.

3. **During runtime, the expanded expressions from the macro call are evaluated**. This part is relatively straightforward — the trick is understanding the above two points as they determine the expression to be evaluated during runtime.

23.4 Macro Walk Through - Three Examples

To walk through the following examples, first start a fresh Lisp session without any predefined global symbols for X. First evaluate the macro definitions to establish the macros, then step through each example - first evaluating macro-expand-1 to observe the macro expansions and then evaluating macro calls for each of the three macros.

	BAD MACRO	GOOD MACRO	BEST MACRO
Macro Definition	`(defmacro bad (x)(+ 1 2 3 x))`	`(defmacro good (x) '(+ 1 2 3 x))`	`(defmacro best (x) `(+ 1 2 3 ,x))`
Example 1: Run the following on a **fresh** Lisp session without X as a predefined symbol			
Macro expansion	`(macroexpand-1 '(bad 8))`	`(macroexpand-1 '(good 8))`	`(macroexpand-1 '(best 8))`
Result of Macro expansion (COMPILE TIME)	Returns 14 Result of evaluating (+ 1 2 3 8). The argument 8 is passed into the macro unevaluated, but the forms in the macro are themselves evaluated. 8 is a literal, hence no error	Returns (+ 1 2 3 X) Quote in the macro definition prevents evaluation of the form during macro expansion	Returns (+ 1 2 3 8) Backquote in the macro definition prevents evaluation of the form during macro expansion. Comma before X forces evaluation of X during READ TIME, hence X is bound to 8 in the macro expansion
Macro Call	`(bad 8)`	`(good 8)`	`(best 8)`
Result of Macro Call (RUN TIME)	Returns 14 This the result of evaluating the value 14 returned by the macro expansion	Error - X is unbound Lisp is trying to evaluate the macroexpansion (+ 1 2 3 X) but is unable to find a value for X as it is unbound	Returns 14 This is the result of the macroexpansion (+ 1 2 3 8)
Example 2: Before the following, first establish a global variable X via `(defparameter x 16)`			
Macro expansion	`(macroexpand-1 '(bad x))`	`(macroexpand-1 '(good x))`	`(macroexpand-1 '(best x))`

Result of Macro expansion (COMPILE TIME)	Error - X is not a Number Result of trying to pass an unevaluated symbol X to the form (+ 1 2 3 X) which gives an error as X is not a number. During macroexpansion, arguments are inserted to the form without evaluation	Returns (+ 1 2 3 X) Quote in the macro definition prevents evaluation of the form during macro expansion	Returns (+ 1 2 3 X) Backquote in the macro definition prevents evaluation of the form during macro expansion However, the comma before X forces evaluation of X during READ TIME, hence the argument X in the macro definition is bound to the symbol X in the macro expansion
Macro Call	`(bad x)`	`(good x)`	`(best x)`
Result of Macro Call (RUN TIME)	Error See above - macro expansion fails in this example	Returns 22 Lisp evaluates the macroexpansion (+ 1 2 3 X) and applies the value of the symbol X here (16)	Returns 22 Lisp evaluates the macro expansion (+ 1 2 3 X) and applies the value of the symbol X here (16).
Example 3: Revisiting example 1, but noting in example 2 we established a global variable X			
Macro expansion	No need to do	`(macroexpand-1 '(good 8))`	`(macroexpand-1 '(best 8))`
Result of Macro expansion (COMPILE TIME)	No need to do	Returns (+ 1 2 3 X) Same as Examples 1 and 2	Returns (+ 1 2 3 8) Same as Example 1
Macro Call	No need to do	`(good 8)`	`(best 8)`
Result of Macro Call (RUN TIME)	No need to do	Returns 22 Same as Example 2. Lisp evaluates the macro expansion (+ 1 2 3 X) , returning 22	Returns 14 Same as Example 1. Argument X is bound to 8 in the macro expansion during READ TIME due to the forced evaluation via ,X

23.5 Additional Uses of Backquote

Comma-at allows us to splice arguments (which should be a list) such that the elements are inserted into the template in place of the list itself:

```
(setf my-list '(a b c))
`(my-list is ,my-list)    ; Returns (MY-LIST IS (A B C))
`(Elements are ,@my-list) ; Returns (ELEMENTS ARE A B C)
```

Comma-at is useful in macros with &rest parameters representing code bodies (note that &body is a synonym and preferable to &rest as it helps some Lisp editors with code formatting):

```
(defmacro my-multi-print-macro (first-line &body body)
`(progn
   (print ,first-line)
   ,@body))

(my-multi-print-macro 55 (print "hello") (print "world"))
```

Finally, we should note that while comma `,` is used for evaluation, a regular quote `'` is used to prevent evaluation within a backquote. Thus, ',(exp) will be converted to (quote exp). In this case, the exp is evaluated to the symbol that is passed into the macro, but the symbol itself is not evaluated as it is quoted (contrast with the second exp in the below):

```
(defmacro test-exp (exp)
 `(format t "~&~S => ~S~%" ',exp ,exp))

(test-exp (+ 5 8))                    ; Returns (+ 5 8) => 13
```

23.6 The Role of Gensym

One issue commonly encountered when writing macros is clashes between identifiers. This can be overcome by utilising the GENSYM function. It creates new and unique symbols each time it is called that are guaranteed not to clash with any pre existing identifiers.

Try the below. There is a lot going on. As an exercise, carefully break it down by applying the rules of backquote and the comma operator we discussed above. You will also need to revisit your understanding of LET forms to fully comprehend the below.

```
(defmacro my-add-macro-2 (x)
  (let ((new-id (gensym)))
    `(let ((,new-id ,x))
       (+ 1 2 3 ,new-id))))

(my-add-macro-2 8)                              ; Returns 14
```

23.7 Macros vs. Functions

Macros are not functions. They cannot be used as functional arguments to functions such as APPLY, FUNCALL or MAP. We can however refer to macros within functions and then use these functions as functional arguments in APPLY, FUNCALL or MAP. The below examples show this, utilising the MY-ADD-MACRO that we defined earlier. Functions can refer to macros within their bodies. The macros will be expanded into full code by the compiler on compilation. We can continue to use APPLY or FUNCALL on such functions.

```
(defun my-function (x)                  ; Define MY-ADD-MACRO
  (my-add-macro x))                     ; before running this

(funcall #'my-function 8)               ; Returns 14
```

The below will not work as we cannot pass macros to a function expecting a function:

```
(funcall #'my-add-macro 8)              ; Error - not a function
```

23.8 Closing Remarks on Macros

It is not always clear when to use macros and when to use functions as they are somewhat similar. One approach is to consider the purpose behind our code:

- When we are writing code to directly solve real world problems, i.e. when our code adds functionality to our applications, a function is likely the best choice.

- When we want to streamline the way we write our code, i.e. by defining shorthand syntax for common blocks of code instructions, macros are likely more appropriate.

Macros extend the language's syntax whilst functions use the language.

24. Further Reading

We have come to the conclusion of this guide and I hope you found it enjoyable and useful. There are quite a few topics that are not covered in this guide, for example CLOS, the condition system and the LOOP macro. Further details on these concepts and also other concepts not covered in this guide can be found in the below resources.

There is much more to learn in Lisp, our purpose today was simply to help you get started in your journey. I really believe Lisp is a beautiful and great language; I hope you do too.

Common Lisp the Language, 2nd Edition
http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

Common Lisp: A Gentle Introduction to Symbolic Computing
http://www.cs.cmu.edu/%7Edst/LispBook/

Practical Common Lisp
http://www.gigamonkeys.com/book/

Common Lisp Cookbook
https://lispcookbook.github.io/cl-cookbook/

Many More Resources (including links to various textbooks) are available on Reddit
https://www.reddit.com/r/lisp/

24.1 Contact Information

Feel free to contact me for any feedback or comments for this guide. If you find it useful, do please star the github repo where it is uploaded, this helps me track the guide's value and also increases its credibility in the eyes of others.

Repo (please star it!)
https://github.com/ashok-khanna/common-lisp-by-example

My e-mail
ashok.khanna@hotmail.com

25. Appendix: Glossary

The below is an extract of the glossary of Common Lisp: A Gentle Introduction to Symbolic Computation by David S. Touretzky.

Address: A number describing the location of an object in memory.

Binding: An archaic term with conflicting uses. Essentially, binding means creating a variable and assigning it a value. See also rebinding.

Block: A named sequence of Lisp expressions, forming the body of a BLOCK expression. Blocks may be exited using RETURN-FROM.

Block name: A symbol serving as the name of a block. DO, DO*, DOTIMES and DOLIST create implicit blocks named NIL. Functions defined by DEFUN or LABELS surround their bodies with implicit blocks whose name is the same as the function.

Body: The body of a form, such as a function definition or a LET, LABELS or DO expression, contains expressions to be evaluated sequentially within the lexical context of the form. Normally, the value of the last expression in the body is returned by the form.

Clause: An element of a COND, AND, or OR conditional expression. A conditional can decide which of its clauses will be evaluated.

Cons cell: The unit of computer memory from which lists are composed. Each cons cell holds two pointers, one in the CAR half, and one in the CDR half.

Dot notation: A notation for writing lists in which cons cells are written as dotted pairs, that is, each cons cell is displayed as a CAR and CDR separated by a dot, enclosed in parentheses. The list (A (B) C) is written (A . ((B . NIL) . (C . NIL))) in dot notation. See also hybrid notation.

Dotted list: A cons cell chain ending in an atom other than NIL. For example, (A B C . D) is a chain of three cons cells ending in the symbol D. This list must be written with a dot to show that the D is the CDR of the third cell, not the CAR of a fourth cell.

Dotted pair: A single cons cell written in do notation. Usually the CAR is a non-NIL atom. A typical dotted pair is (A . B).

Element: The elements of a list are the cars of its top-level cons cells, that is, the things that appear within only one level of parentheses.

Function: Functions transform inputs to outputs. Lisp functions are defined with DEFUN. Lisp programs are organized as collections of functions.

Function cell: One of the five components of a symbol. The function cell holds a pointer to the function object representing the global function named by that symbol. (Local functions created by LABELS do not reside in the function cell.)

Function object: A piece of Lisp data that is a function, and can be applied to arguments. The representation of function objects is implementation dependent.

Gensym: A symbol created automatically, with a name such as #:G0037, that is not registered in any package. Gensyms are often found in the expansions of complex macros such as SETF.

Lambda: A marker indicating that a list is a lambda expression and is to be interpreted as a description of a function.

Lambda-list keyword: A special symbol such as &OPTIONAL or &REST that has a special meaning when it appears in the argument list of a function.

Lambda expression: A list that describes a function. Its first element must be the symbol LAMBDA, its second element must be an argument list, and its remaining elements constitute the body of the function. Lambda expressions must be quoted with #'. For example, #'(LAMBDA (N) (* N 2)).

Lexical closure: A type of function. Lexical closures are created automatically by Lisp when functions passed as arguments to other functions need to remember their lexical context.

Lexical scoping: A scoping discipline in which the only variables a function can see are those it defined itself, plus those defined by forms that contain the function, as when a function defined with DEFUN contains a lambda expression inside it.

List: A chain of cons cells. One of the fundamental data structures of Lisp.

Macro function: A special kind of function whose arguments are not evaluated. Macro functions must return Lisp expressions, which are then evaluated.

Package: Packages are the name spaces in which symbols are registered. The default package is called USER. Lisp functions and variables are named by symbols in package Lisp.

Package name: A character string giving the name of a package, such as USER. APROPOS takes a package name as an optional second argument.

Pointer: A pointer to an object gives the address of that object in memory. Pointers are drawn as arrows in cons cell diagrams.

Primitive: An elementary function that is built into Lisp, not defined by the user. CONS and + are primitives.

Proper list: A cons cell chain ending in NIL. NIL is itself a proper list.

Rebinding: Rebinding a special variable means creating a new dynamic variable with the same name, such as with LET. The name is then dynamically associated with the new variable when it appears anywhere in the program, and the old variable is inaccessible until the form that bound the new variable returns.

Scope: The scope of an object is the region of the program in which the object can be referenced. For example, if a variable names the input to some function, the scope of the variable is limited to the body of that function. See also lexical scoping and dynamic scoping.

Special form: See special function.

Special function: A built-in function that does not evaluate its arguments. Special functions provide the primitive constructs, such as assignment, block structure, looping, and variable binding, from which the rest of Lisp is built. They do not return Lisp expressions to be evaluated, as macros do. Lisp programmers can create new macros, but they cannot create new special functions.

Special variable: A dynamically scoped variable. When a name is declared special, all variables with that name will be dynamically scoped.

String: A sequence of characters enclosed in double quotes, e.g. "Foo Bar". Strings are vectors of character objects.

Symbol: One of the fundamental Lisp datatypes. Internally, symbols are composed of five cells: the name, value, function, plist, and package cells. Besides serving as data, symbols also serve as names for things, such as functions, variables, types, and blocks.

Symbol name: Symbols are named by character strings. Each symbol contains a name cell that holds a pointer to the character string that is the symbol's name.

Type system: The set of datatypes a language offers, and their organization. The Lisp type system includes type predicates, a TYPE-OF function for generating type descriptions, and a facility for creating new datatypes with DEFSTRUCT.

Unassigned variable: A variable that has no value.

Unbound variable: See unassigned variable. ''Unbound'' is an archaic term for ''unassigned''.

Value cell: A cell in the internal representation of a symbol where Lisp keeps the value of the global lexical variable (or the currently accessible dynamic variable) named by that symbol.

Variable: A place where a value is stored. Ordinary variables are named by symbols. Generalized variables are named by place descriptions, which may be Lisp expressions.

Vector: A one-dimensional array.