Elisp

Mingming Li

First Created: March 29, 2022
Last Modified: December 17, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most of the GNU macs text editor is written in the programming language called Emacs Lisp. It is useful to do **customization** and **extension** to Emacs.

1.1 History

Lisp (LISt Processing language) was first developed in the late 1950s at the Massachusetts Institute of Technology for research in artificial intelligence.

Dozens of Lisp implementations have been built over the years, each with its own idiosyncrasies. Many of them were inspired by Maclisp, which was written in the 1960s at MIT's Project MAC. Eventually the implementers of the descendants of Maclisp came together and developed a standard for Lisp systems, called Common Lisp.

GNU Emacs Lisp is largely inspired by Maclisp, and a little by Common Lisp.

Chapter 2

Lisp Data Types

A Lisp **object** is a piece of data used and manipulated by Lisp programs. A **type** or **data type** is a set of possible objects.

Every object belongs to at least one type. Objects of the same type have similar structures and may usually be used in the same contexts. Types can overlap, and objects can belong to two or more types. Consequently, we can ask whether an object belongs to a particular type, but not for the type of an object. (This may differ from other programming language)

A few fundamental object types are built into Emacs. These, from which all other types are constructed, are called **primitive types**. Each object belongs to **one and only one** primitive type. These types include **integer, float, cons, symbol, string, vector, hash-table, subr, byte-code function**, and **record**, plus several special types, such as **buffer**, that are related to editing.

Each primitive type has a corresponding Lisp function that checks whether an object is a member of that type.

Lisp is unlike many other languages in that its objects are **self-typing**: the primitive type of each object is **implicit** in the object itself. For example, if an object is a vector, nothing can treat it as a number; Lisp knows it is a vector, not a number.

In most languages (like Java), the programmer must declare the data type of each variable, and the type is known by the compiler but not represented in the data. Such type declarations do not exist in Emacs Lisp. A Lisp variable can have any type of value, and it remembers whatever value you store in it, type and all.

2.1 Printed Representation and Read Syntax

The **printed representation** of an object is the format of the output generated by the Lisp printer (the function **prin1**) for that object. Every data type has a **unique** printed representation. The **read syntax** of an object is the format of the input accepted by the Lisp reader (the function **read**) for that object. This is not necessarily unique; many kinds of object have more than one syntax.

In most cases, an object's printed representation is also a read syntax for the object.

In other languages, an expression is text; it has no other form. In Lisp, an expression is primarily a Lisp object and only secondarily the text that is the object's read syntax.

2.2 Special Read Syntax

Emacs Lisp represents many special objects and constructs via special hash notations.

- **#<...>**
  Objects that have no read syntax are presented like this.
- **##**
  The printed representation of an interned symbol whose name is an empty string.
- **#'**
  This is a shortcut for **function**.
- **#:**
  The printed representation of an uninterned symbol whose name is `foo` is `#:foo`.
- **#N**
  When printing circular structures, this construct is used to represent where the structure loops back onto itself, and 'N' is the starting list count:

```
(let ((a (list 1)))
  (setcdr a a))
;; => (1 . #0)
```

- **#N=**
- **#N#**
  #N= gives the name to an object and **#N#** represents that object, so when reading back the object, they will be the same object instead of copies.
- **#xN**
  'n' represented as a hexadecimal number (`#x2a`).
- **#oN**
  'N' represented as an octal number (`#o52`).
- **#bN**
  'N' represented as a binary number (`#b101010`).
- **#(...)**
  String text properties.
- **#^**
  A char table.
- **#s(hash-table ...)**
  A hash table.
- **?C**
  A character.

- **#$**
  The current file name in byte-compiled files.
- **#@N**
  Skip the next ‘N’ characters.
- **#f**
  Indicates that the following form isn’t readable by the Emacs Lisp reader. This is only in text for display purposes and will never appear in any Lisp file.

2.3 Comments

A **comment** is text that is written in a program only for the sake of humans that read the program, and that has no effect on the meaning of the program. In Lisp, an unescaped semicolon (**;**) starts a comment if it is not within a string or character constant. The comment continues to the end of line.

2.4 Programming Types

There are two general categories of types in Emacs Lisp:

- those having to do with Lisp programming. (exist in many Lisp implementation.)
- those having to do with editing. (unique to Emacs Lisp.)

Type in Emacs Lisp is like class in structure in C, class in Java and Python.

2.4.1 Integer Type

There are two kinds of integers:

- small integers, called **fixnums**
- large integers, called **bignums**.

The range of values for a fixnum depends on the machine. The minimum range is -2^{29} to $2^{29} - 1$ but many machines provide a wider range. Bignums can have arbitrary precision. Operations that overflow a fixnum will return a bignum instead.

All numbers can be compared with `eql` or `=`; fixnums can also be compared with `eq`. To test whether an integer is a fixnum or a bignum, you can use predicates `fixnump` and `bignump`.

The read syntax for integers is a sequence of (base ten) digits with an optional sign at the beginning and an optional period at the end. The printed representation produced by the Lisp interpreter never has a leading ‘+’ or a final ‘.’.

```
-1   ; -1
1    ; 1
1.   ; 1
+1   ; 1
```

2.4.2 Floating-Point Type

Emacs uses the C data type `double` to store the floating-point value.

The printed representation for floating-point numbers requires either a decimal point (with at least one digit following), an exponent, or both.

```
;; 1500
1500.0
+15e2
15.0e+2
+1500000e-3
.15e4
```

2.4.3 Character Type

A character in Emacs Lisp is nothing more than an integer. In other words, characters are represented by their character codes. For example, the character A is represented as the integer 65.

Basic Char Syntax

Since characters are really integers, the printed representation of a character is a decimal number. This is also a possible read syntax for a character, but writing characters that way in Lisp programs is not clear programming. You should **always** use the special read syntax formats that Emacs Lisp provides for characters. These syntax formats start with a question mark.

The usual read syntax for alphanumeric characters is a question mark followed by the character; thus, '?A' for the character A, '?B' for the character B, and '?a' for the character a.

```
?A                                      ; 65
?B                                      ; 66
?a                                      ; 97
```

You can use the same syntax for punctuation characters. However, if the punctuation character has a special syntactic meaning in Lisp, you must quote it with a '\'. For example, '`?\(`' is the way to write the open-paren character. Likewise, if the character is '\', you must use a second '\' to quote it: '?\\'.

```

?\a                                     ; control-g
?\b                                     ; backspace, BS
?\t                                     ; tab, TAB
?\n                                     ; newline
?\\                                     ; backslash character, \
?\d                                     ; delete character, DEL
?\e                                     ; escape character, ESC
?\r                                     ; carriage return, RET
```

These sequences which start with backslash are also known as **escape sequences**, because backslash plays the role of an escape character. A backslash is allowed, and harmless, preceding any character without a special escape meaning; thus, '?\+' is equivalent to '?+'.

General Escape Syntax

In addition to the specific escape sequences for special important control characters, Emacs provides several types of escape syntax that you can use to specify non-ASCII text characters.

- You can specify characters by their Unicode names, if any. ?\{NAME} represents the Unicode character named NAME. `?\N{LATIN SMALL LETTER A WITH GRAVE}`.
- You can specify characters by their Unicode values. `?\N{U+X}` represents a character with Unicode code point X, where X is a hexadecimal number. `?\N{U+E0}`.
- You can specify characters by their hexadecimal character codes. A hexadecimal escape sequence consists of a backslash, 'x', and the hexadecimal character code. `?\x41`.
- You can specify characters by their character code in octal. An octal escape sequence consists of a backslash followed by up to three octal digits. `?\101`.

Control-Character Syntax

Control characters can be represented using yet another read syntax. This consists of a question mark (**?**) followed by a backslash (\), caret (**^**), and the corresponding non-control character, in either upper or lower case. Instead of the **^**, you can use **C-**.

```
?\^I                                    ; C-i
?\^i                                    ; C-i
?\C-I                                   ; C-i
?\C-i                                   ; C-i
```

Meta-Character Syntax

A **meta** character is a character typed with the META modifier key. The read syntax for meta characters uses **\\M-**.

```
?\M-A                                   ; M-A
?\M-\C-b                                ; C-M-b
?\C-\M-b                                ; C-M-b
```

Other Character Modifier Bits

The case of a graphic character is indicated by its character code; for example, ASCII distinguishes between the characters 'a' and 'A'. But ASCII has no way to represent whether a control character is upper case or lower case. Emacs uses the 25th bit (2^{25}) to indicate that the shift key was used in typing a control character. This distinction is possible only on a graphical display such as a GUI display on X; text terminals do not report the distinction. The X Window System defines three other modifier bits that can be set in a character: hyper, super and alt. The bit values are 22th (2^{22}) for alt, 23th 2^{23} for super and 24th 2^{24} for hyper.

```
\S-                                     ; shift
\H-                                     ; hyper
\s-                                     ; super
\A-                                     ; alt
```

2.4.4 Symbol Type

A **symbol** in GNU Emacs Lisp is an object with a name. The symbol name serves as the printed representation of the symbol. In ordinary Lisp use, with one single obarray, a symbol's name is unique --no two symbols have the same name.

A symbol can serve as a variable, as a function name, or to hold a property list. Or it may serve only to be distinct from all other Lisp objects, so that its presence in a data structure may be recognized reliably. In a given context, usually only one of these uses is intended. But you can use one symbol in all of these ways, independently.

A symbol whose name starts with a colon (':') is called a keyword symbol. These symbols automatically act as constants, and are normally used only by comparing an unknown symbol with a few specific alternatives.

A symbol name can contain any characters whatever. If a symbol name looks like a number, you need to write a "\" at the beginning of the name to force interpretation as a symbol. The characters `-+=*/_~!@$%^&:<>{}?` requires no special punctuation. Any other characters my be included in a symbol's name by escaping them with a backslash. In contract to its use in string, a backslash in the name of a symbol simply quotes the single character that follows the backslash. For example, in a string, '\t' represents a tab character; in the name of a symbol, however, '\t' merely quotes the letter 't'. To have a symbol with a tab character in its name, you must actually use a tab (preceded with a backslash). But it's rare to do such a thing.

```
foo                                         ; A symbol named 'foo'
FOO                                         ; A symbol named 'FOO'
1+                                          ; A symbol named '1+'
\+1                                         ; A symbol named '+1'
+-*/_~!@$%^&=:<>{}                          ; A symbol named '+-*/_~!@$%^&=:<>{}'
```

As an exception to the rule that a symbol's name serves as its printed representation, '##' is the printed representation for an interned symbol whose name is an empty string. Furthermore, '#:foo' is the printed representation for an uninterned symbol whose name is foo.

2.4.5 Sequence Types

A **sequence** is a Lisp object that represents an ordered set of elements. There are two kinds of sequence in Emacs Lisp: lists and arrays.

A list can hold elements of any type, and its length can be easily changed by adding or removing elements.

Arrays are fixed-length sequences. They are further subdivided into:

- strings
  Elements can only be characters.
- vectors
  Elements can by any types.
- char-tables
  Like vectors except that they are indexed by any valid character code.
- bool-vectors
  Elements must be `t` or `nil`.

It is generally impossible to read the same sequence twice, since sequences are always created a new upon reading. If you read the read syntax for a sequence twice, you get two sequences with equal contents. There is one exception: the empty list () always stands for the same object, `nil`.

2.4.6 Cons Cell and List Types

A **cons cell** is an object that consists of two slots, called the **car** slot and the **cdr** slot. Each slot can hold any Lisp object.

A list is a series of cons cells, linked together so that the cdr slot of each cons cell holds either the next cons cell or the empty list. The empty list is actually the symbol nil. Because most cons cells are used as part of lists, we refer to any structure made out of cons cells as a **list structure**.

Because cons cells are so central to Lisp, we also have a word for an object which is not a cons cell. These objects are called **atoms**.

The read syntax and printed representation for lists are identical, and consist of a left parenthesis, an arbitrary number of elements, and a right parenthesis.

```
(A 2 "A")            ; A list of three elements.
()                   ; A list of no elements (the empty list).
nil                  ; A list of no elements (the empty list).
("A ()")             ; A list of one element: the string "A ()".
(A ())               ; A list of two elements: A and the empty list.
(A nil)              ; Equivalent to the previous.
((A B C))            ; A list of one element (which is a list of three elements).
```

Upon reading, each object inside the parentheses becomes an element of the list. That is, a cons cell is made for each element. The car slot of the cons cell holds the element, and its cdr slot refers to the next cons cell of the list, which holds the next element in the list. The cdr slot of the last cons cell is set to hold `nil`.

The names car and cdr derive from the history of Lisp. The original Lisp implementation ran on an IBM 704 computer which divided words into two parts, the address and the decrement; car was an instruction to extract the contents of the address part of a register, and cdr an instruction to extract the contents of the decrement. By contrast, cons cells are named for the function **cons** that creates them, which in turn was named for its purpose, the construction of cells.

Dotted Pair Notation

Dotted pair notation is a general syntax for cons cells that represents the car and cdr explicitly. In this syntax, `(a . b)` stands for a cons cell whose car is the object `a` and whose cdr is the object `b`. It has the advantage that the cdr does not have to be a list. However, it is more cumbersome in cases where list syntax would work. In dotted pair notation, the list `(1 2 3)` is written as `(1 . (2 . (3 . nil)))`. When printing a list, the dotted pair notation is only used if the cdr of a cons cell is not a list.

Association List Type

An **association list** or **alist** is a specially-constructed list whose elements are cons cells. In each element, the car is considered a **key**, and the cdr is considered an **associated value**.

```
(setq alist-of-colors
      '((rose . red) (lily . white) (buttercup . yellow)))
```

2.4.7 Array Type

An **array** is composed of an arbitrary number of slots for holding or referring to other Lisp objects, arranged in a contiguous block of memory. (like array in data structure)

2.4.8 String Type

Strings in Lisp are constants: evaluation of a string returns the same string.

Syntax for Strings

The read syntax for a string is a double-quote, an arbitrary number of characters, and another double-quote (`"like this."`). To include a double-quote in a string, precede it with a backslash (`"\""`). Likewise, you can include a backslash by preceding it with another backslash (`"\\"`).

The newline character is not special in the read syntax for strings; if you write a new line between the double-quotes, it becomes a character in the string. But an escaped newline — one that is preceded by '\'—does not become part of the string; i.e., the Lisp reader ignores an escaped newline while reading a string. An escaped space '\ ' is likewise ignored.

Non-ASCII Characters in Strings

There are two text representations for non-ASCII characters in Emacs strings:

- multibyte
  Its value maybe between 0 and 4194303 ($2^{22} - 1$). It store raw bytes.
- unibyte
  Its value is between 0 and 255 ($2^{8} - 1$). It store human-readable text.

In both cases, characters above 127 are non-ASCII.

You can include a non-ASCII character in a string constant by writing it literally. Instead of writing a character literally into a multibyte string, you can write it as its character code using an escape sequence.

Nonprinting Characters in Strings

You can use the same backslash escape-sequences in a string constant as in character literals (but do not use the question mark that begins a character constant).

```
"\t"                                    ; tab
"\C-a"                                  ; C-a
```

However, not all of the characters you can write with backslash escape-sequences are valid in strings. The only control characters that a string can hold are the ASCII control characters. Strings do not distinguish case in ASCII control characters.

Properly speaking, strings cannot hold meta characters; but when a string is to be used as a key sequence, there is a special convention that provides a way to represent meta versions of ASCII characters in a string. If you use the '\M-' syntax to indicate a meta character in a string constant, this sets the 2^7 bit of the character in the string. If the string is used in `define-key` or `lookup-key`, this numeric code is translated into the equivalent meta character.

Strings cannot hold characters that have the hyper, super, or alt modifiers.

Text Properties in Strings

A string can hold properties for the characters it contains, in addition to the characters themselves. This enables programs that copy text between strings and buffers to copy the text's properties with no special effort. Strings with text properties use a special read and print syntax:

```
#("characters" property-data...)
```

where `property-data` consists of zero or more elements, in groups of three as follows:

```
beg end plist
```

The elements `beg` and `end` are integers, and together specify a range of indices in the string; `plist` is the property list for that range. For example:

```
#("foo bar" 0 3 (face bold) 3 4 nil 4 7 (face italic))
```

2.4.9 Vector Type

The printed representation of a vector consists of a left square bracket, the elements, and a right square bracket. This is also the read syntax. Like numbers and strings, vectors are considered constants for evaluation.

```
[1 "two" (three)]                       ; A vector of three elements.
```

2.4.10 Char-Table Type

Char-tables have certain extra features to make them more useful for many jobs that involve assigning information to character codes — for example, a char-table can have a parent to inherit from, a default value, and a small number of extra slots to use for special purposes. A char-table can also specify a single value for a whole character set.

The printed representation of a char-table is like a vector except that there is an extra `#^` at the beginning.

2.4.11 Bool-Vector Type

The printed representation of a bool-vector is like a string, except that it begins with `#&` followed by the length. The string constant that follows actually specifies the contents of the bool-vector as a bitmap—each character in the string contains 8 bits, which specify the next 8 elements of the bool-vector (1 stands for t, and 0 for nil). The least significant bits of the character correspond to the lowest indices in the bool-vector.

```
(make-bool-vector 3 t)                  ; #&3"^G"
(make-bool-vector 3 nil)                ; #&3"^@"
```

These results make sense, because the binary code for 'C-g' is 111 and 'C-@' is the character with code 0.

If the length is not a multiple of 8, the printed representation shows extra elements, but these extras really make no difference. For instance, in the next example, the two bool-vectors are equal, because only the first 3 bits are used:

```
(equal #&3"\377" #&3"\007")             ; t
```

2.4.12 Hash Table Type

The printed representation of a hash table specifies its properties and contents, like this:

```
(make-hash-table)
;; #s(hash-table size 65 test eql rehash-size 1.5 rehash-threshold 0.8125 data (
    ...))
```

2.4.13 Function Type

Lisp functions are executable code. Functions are also Lisp objects. A non-compiled function is a lambda expression: that is, a list whose first element is the symbol `lambda`.

In Lisp, a function has no intrinsic name. A lambda expression can be called as a function even though it has no name; to emphasize this, we also call it an **anonymous function**. A named function in Lisp is just a symbol with a valid function in its function cell.

Most of the time, functions are called when their names are written in Lisp expressions in Lisp programs. However, you can construct or obtain a function object at run time and then call it with the primitive functions `funcall` and `apply`.

2.4.14 Macro Type

A **Lisp macro** is a user-defined construct that extends the Lisp language. It is represented as an object much like a function, but with different argument-passing semantics. A Lisp macro has the form of a list whose first element is the symbol `macro` and whose cdr is a Lisp function object, including the `lambda` symbol.

Lisp macro objects are usually defined with the built-in `defmacro` macro, but any list that begins with `macro` is a macro as far as Emacs is concerned.

2.4.15 Primitive Function Type

A **primitive function** is a function callable from Lisp but written in the C programming language. Primitive functions are also called **subrs** or **built-in functions**. (The word “subr” is derived from “subroutine”.) Most primitive functions evaluate all their arguments when they are called. A primitive function that does not evaluate all its arguments is called a **special form**.

Primitive functions have no read syntax and print in hash notation with the name of the subroutine.

```
;; symbol-function: access the function cell of the symbol
(symbol-function 'car)                ; #<subr car>
;; To check if it is a primitive function.
(subrp (symbol-function 'car))        ; t
```

2.4.16 Byte-Code Function Type

Byte-code function objects are produced by byte-compiling Lisp code. Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears in a function call. The printed representation and read syntax for a byte-code function object is like that for a vector, with an additional `#` before the opening `[` (i.e. start with `#[`).

2.4.17 Record Type

A **record** is much like a `vector`. However, the first element is used to hold its type as returned by `type-of`. The purpose of records is to allow programmers to create objects with new types that are not built into Emacs.

2.4.18 Type Descriptors

A **type descriptor** is a `record` which holds information about a type. Slot 1 in the record must be a symbol naming the type, and `type-of` relies on this to return the type of `record` objects. No other type descriptor slot is used by Emacs; they are free for use by Lisp extensions.

2.4.19 Autoload Type

An **autoload object** is a list whose first element is the symbol `autoload`. It is stored as the function definition of a symbol, where it serves as a placeholder for the real definition. The autoload object says that the real definition is found in a file of Lisp code that should be loaded when necessary. It contains the name of the file, plus some other information about the real definition.

After the file has been loaded, the symbol should have a new function definition that is not an autoload object. The new definition is then called as if it had been there to begin with. From the user's point of view, the function call works as expected, using the function definition in the loaded file.

2.4.20 Finalizer Type

A **finalizer object** helps Lisp code clean up after objects that are no longer needed. A finalizer holds a Lisp function object. When a finalizer object becomes unreachable after a garbage collection pass, Emacs calls the finalizer's associated function object. When deciding whether a finalizer is reachable, Emacs does not count references from finalizer objects themselves, allowing you to use finalizers without having to worry about accidentally capturing references to finalized objects themselves.

Errors in finalizers are printed to `*Messages*`. Emacs runs a given finalizer object's associated function exactly once, even if that function fails.

```
(make-finalizer function)
```

Make a finalizer that will run `function`. `function` will be called after garbage collection when the returned finalizer object becomes unreachable. If the finalizer object is reachable only through references from finalizer objects, it does not count as reachable for the purpose of deciding whether to run `function`. `function` will be run once per finalizer object.

2.5 Editing Types

2.5.1 Buffer Type

A **buffer** is an object that holds text that can be edited. Most buffers hold the contents of a disk file so they can be edited, but some are used for other purposes (like *scratch* buffer). Most buffers are also meant to be seen by the user, and therefore displayed, at some time, in a window. But a buffer need not be displayed in any window. Each buffer has a designated position called point; most editing commands act on the contents of the current buffer in the neighborhood of point. At any time, one buffer is the **current buffer**.

Many of the standard Emacs functions manipulate or test the characters in the current buffer. Several other data structures are associated with each buffer:

- a local syntax table
- a local keymap
- a list of buffer-local variable bindings
- overlays
- text properties for the text in the buffer

The local keymap and variable list contain entries that individually override global bindings or values. These are used to customize the behavior of programs in different buffers, without actually changing the programs.

Buffers have no read syntax. They print in hash notation, showing the buffer name.

```
(current-buffer)                          ; #<buffer *scratch*>
```

A **marker** denotes a position in a specific buffer. Markers therefore have two components: one for the buffer, and one for the position. Changes in the buffer's text automatically relocate the position value as necessary to ensure that the marker always points between the same two characters in the buffer.

Markers have no read syntax. They print in hash notation, giving the current character position and the name of the buffer.

```
(point-marker)                            ; #<marker at 3035 in *scratch*>
```

2.5.2 Window Type

A **window** describes the portion of the screen that Emacs uses to display buffers. Every live window has one associated buffer, whose contents appear in that window. By contrast, a given buffer may appear in one window, no window, or several windows. Windows are grouped on the screen into **frames**; each window belongs to one and only one frame.

Though many windows may exist simultaneously, at any time one window is designated the selected window. This is the window where the cursor is (usually) displayed when Emacs is ready for a command. The selected window usually displays the current buffer, but this is not necessarily the case.

Windows have no read syntax. They print in hash notation, giving the window number and the name of the buffer being displayed. The window numbers exist to identify windows uniquely, since the buffer displayed in any given window can change frequently.

```
(selected-window)                         ; #<window 3 on *scratch*>
```

2.5.3 Frame Type

A **frame** is a screen area that contains one or more Emacs windows; we also use the term **frame** to refer to the Lisp object that Emacs uses to refer to the screen area.

Frames have no read syntax. They print in hash notation, giving the frame's title, plus its address in core (useful to identify the frame uniquely).

```
(selected-frame)                         ; #<frame *scratch* 0x7fdcdc0e2630>
```

2.5.4 Termial Type

A **terminal** is a device capable of displaying one or more Emacs frames.

Terminals have no read syntax. They print in hash notation giving the terminal's ordinal number and its TTY device file name.

```
;; on my macbook
(get-device-terminal nil)                ; #<terminal 1 on Mingmings-MacBook-Pro.
    local>

;; on linux
(get-device-terminal nil)                ; #<terminal 1 on /dev/tty>
```

2.5.5 Window Configuration Type

A **window configuration** stores information about the positions, sizes, and contents of the windows in a frame, so you can recreate the same arrangement of windows later.

Window configurations do not have a read syntax; their print syntax looks like `#<window-configuration>`.

2.5.6 Frame Configuration Type

A **frame configuration** stores information about the positions, sizes, and contents of the windows in all frames. It is not a primitive type. It is actually a list whose car is `frame-configuration` and whose cdr is an alist. Each alist element describes one frame, which appears as the car of that element.

2.5.7 Process Type

The word **process** usually means a running program. Emacs itself runs in a process of this sort. However, in Emacs Lisp, a process is a Lisp object that designates a subprocess created by the Emacs process. Programs such as shells, GDB, ftp, and compilers, running in subprocesses of Emacs, extend the capabilities of Emacs. An Emacs subprocess takes textual input from Emacs and returns textual output to Emacs for further manipulation. Emacs can also send signals to the subprocess.

Process objects have no read syntax. They print in hash notation, giving the name of the process:

```
(process-list)                           ; (#<process shell> #<process server> #<
    process ispell>)
```

2.5.8 Thread Type

A **thread** in Emacs represents a separate thread of Emacs Lisp execution. It runs its own Lisp program, has its own current buffer, and can have subprocesses locked to it, i.e. subprocesses whose output only this thread can accept.

Thread objects have no read syntax. They print in hash notation, giving the name of the thread (if it has been given a name) or its address in core:

```
(all-threads)                            ; (#<thread 0x10b588b80>)
```

2.5.9 Mutex Type

A **mutex** is an exclusive lock that threads can own and disown, in order to synchronize between them.

Mutex objects have no read syntax. They print in hash notation, giving the name of the mutex (if it has been given a name) or its address in core:

```
(make-mutex "my-mutex")                 ; #<mutex my-mutex>
(make-mutex)                            ; #<mutex 0x7fdcdf9deb60>
```

2.5.10 Condition Variable Type

A **condition variable** is a device for a more complex thread synchronization than the one supported by a mutex. A thread can wait on a condition variable, to be woken up when some other thread notifies the condition.

Condition variable objects have no read syntax. They print in hash notation, giving the name of the condition variable (if it has been given a name) or its address in core:

```
(make-condition-variable (make-mutex))  ; #<condvar 0x7fdcdfd33980>
```

2.5.11 Stream Type

A **stream** is an object that can be used as a source or sink for characters — either to supply characters for input or to accept them as output. Many different types can be used this way: markers, buffers, strings, and functions. Most often, input streams (character sources) obtain characters from the keyboard, a buffer, or a file, and output streams (character sinks) send characters to a buffer, such as a *Help* buffer, or to the echo area.

The object `nil`, in addition to its other meanings, may be used as a stream. It stands for the value of the variable `standard-input` or `standard-output`. Also, the object `t` as a stream specifies input using the minibuffer or output in the echo area.

Streams have no special printed representation or read syntax, and print as whatever primitive type they are.

2.5.12 Keymap Type

A **keymap** maps keys typed by the user to commands. This mapping controls how the user's command input is executed. A keymap is actually a list whose car is the symbol `keymap`.

```
(symbol-value 'lisp-mode-map)
;; (keymap
;;   (3 keymap
;;      ;; C-c C-z
;;      (26 . run-lisp))
;;   (27 keymap
;;       ;; C-M-x, treated as ESC C-x
;;       (24 . lisp-send-defun))
;;   ;; This part is inherited from lisp-mode-shared-map. keymap
;;   ;; DEL
;;   (127 . backward-delete-char-untabify)
;;   (27 keymap
;;       ;; C-M-q, treated as ESC C-q
;;       (17 . indent-sexp)))
```

2.5.13 Overlay Type

An **overlay** specifies properties that apply to a part of a buffer. Each overlay applies to a specified range of the buffer, and contains a property list (a list whose elements are alternating property names and values). Overlay properties are used to present parts of the buffer temporarily in a different display style. (narrowing)

Overlays have no read syntax, and print in hash notation, giving the buffer name and range of positions.

```
(setq foo (make-overlay 1 10)) ; #<overlay from 1 to 10 in scratch.el>
```

2.5.14 Font Type

A **font** specifies how to display text on a graphical terminal. There are actually three separate font types — **font objects**, **font specs**, and **font entities** — each of which has slightly different properties. None of them have a read syntax; their print syntax looks like `#<font-object>`, `#<font-spec>`, and `#<font-entity>` respectively.

2.6 Read Syntax for Circular Objects

To represent shared or circular structures within a complex of Lisp objects, you can use the reader constructs `#n=` and `#n#`.

Use `#n=` before an object to label it for later reference; subsequently, you can use `#n#` to refer the same object in another place. Here, `n` is some integer.

```
;; make a list in which the first element recurs as the third element
(#1=(a) b #1#)

;; This differs from ordinary syntax such as this
((a) b (a))

(prog1 nil
  (setq x '(#1=(a) b #1#)))
(eq (nth 0 x) (nth 2 x))                ; t

(setq x '((a) b (a)))
(eq (nth 0 x) (nth 2 x))                ; nil
```

You can also use the same syntax to make a circular structure, which appears as an element within itself.

```
;; This makes a list whose second element is the list itself.
#1=(a #1#)
```

2.7 Type Predicates

The Emacs Lisp interpreter itself does not perform type checking on the actual arguments passed to functions when they are called. It could not do so, since function arguments in Lisp do not have declared data types, as they do in other programming languages. It is therefore up to the individual function to test whether each actual argument belongs to a type that the function can use.

All built-in functions do check the types of their actual arguments when appropriate, and signal a wrong-type-argument error if an argument is of the wrong type.

If you want your program to handle different types differently, you must do explicit type checking. The most common way to check the type of an object is to call a **type predicate** function. Emacs has a type predicate for each type, as well as some predicates for combinations of types.

A type predicate function takes one argument; it returns t if the argument belongs to the appropriate type, and nil otherwise. Following a general Lisp convention for predicate functions, most type predicates' names end with p.

Here is a table of predefined type predicates, in alphabetical order:

- atom
- arrayp
- bignump
- bool-vector-p
- booleanp
- bufferp
- byte-code–function-p
- case-table-p
- char-or-string-p
- char-table-p
- commandp
- condition-variable-p
- consp
- custom-variable-p
- fixnump
- floatp
- fontp
- frame-configuration-p
- frame-live-p
- framep
- functionp
- hash-table-p
- integer-or-marker-p

- integerp
- keymapp
- keywordp
- listp
- markerp
- mutexp
- nlistp
- number-or-marker-p
- numberp
- overlayp
- processp
- recordp
- sequencep
- string-or-null-p
- stringp
- subrp
- symbolp
- syntax-table-p
- treadp
- vectorp
- wholenump
- window-configuration-p
- window-live-p
- windowp

The most general way to check the type of an object is to call the function `type-of`. Each object belongs to one and only one primitive type; `type-of` tells you which one. But `type-of` knows nothing about non-primitive types. In most cases, it is more convenient to use type predicates than `type-of`.

2.8 Equality Predicates

Here we describe functions that test for equality between two objects.

♥ `(eq object1 object2)`
This function returns `t` if `object1` and `object2` are the same object, and `nil` otherwise.
- If `object1` and `object2` are symbols with the same name, they are normally the same object (there are exceptions[1]).
- For other non-numeric types (e.g., lists, vectors, strings), two arguments with the same contents or elements are not necessarily `eq` to each other: they are `eq` only if they are the same object, meaning that a change in the contents of one will be reflected by the same change in the contents of the other.
- If `object1` and `object2` are numbers with differing types or values, then they cannot be the same object and `eq` returns `nil`.
- If they are fixnums with the same value, then they are the same object and `eq` returns `t`.
- If they were computed separately but happen to have the same value and the same non-fixnum numeric type, then they might or might not be the same object, and `eq` returns `t` or `nil` depending on whether the Lisp interpreter created one object or two.

♥ `(equal object1 object2)`
This function returns t if object1 and object2 have equal components, and nil otherwise.
Comparison of strings is case-sensitive, but does not take account of text properties — it compares only the characters in the strings.
The `equal` function recursively compares the contents of objects if they are inte gers, strings, markers, vectors, bool-vectors, byte-code function objects, char-tables, records, or font objects. Other objects are considered `equal` only if they are `eq`. For example, two distinct buffers are never considered equal, even if their textual contents are the same.

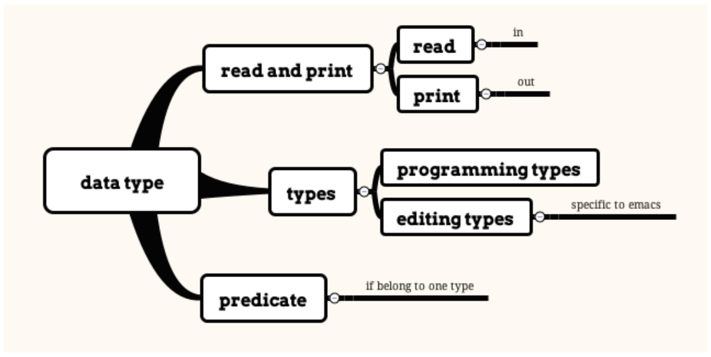
♥ `(equal-including-properties object1 object2)`
This function behaves like `equal` in all cases but also requires that for two strings to be equal, they have the same text properties.

2.9 Mutability

Some Lisp object should or can never change. But some Lisp objects can change, we say they are **mutable**.

A mutable object stops being mutable if it is part of expression that is evaluated.

2.10 Summary



Figures 2.1: List data types

[1]The `make-symbol` function returns an uninterned symbol, distinct from the symbol that is used if you write the name in a Lisp expression. Distinct symbols with the same name are not `eq`.

Chapter 3

Numbers

GNU Emacs support two numeric data types: **integers** and **floating-point numbers**.

3.1 Iteger Basics

The Lisp reader reads an integer as nonempty sequence of decimal digits with optional initial sign and optional final period.

```
1
1.
+1
-1
0
-0
```

The syntax for integers in bases other than 10 consists of # followed by a radix indication followed by one or more digits.

```
;; binary
#b101100                              ; 44
;; octal
#o54                                  ; 44
;; hex
#x2c                                  ; 44
;; #radix r integer, radix=24
#24r1k                                ; 44
```

Many of the functions described in this chapter accept markers for arguments in place of numbers. Since the actual arguments to such functions may be either numbers or markers, we often give these arguments the name `number-or-marker`. When the argument value is a marker, its position value is used and its buffer is ignored.

In Emacs Lisp, text characters are represented by integers. Any integer between zero and the value of `(max-char)`, inclusive, is considered to be valid as a character.

Integers in Emacs Lisp are not limited to the machine word size. Under the hood, though, there are two kinds of integers: smaller ones, called **fixnums**, and larger ones, called **bignums**.

```
most-positive-fixnum                    ; 2305843009213693951
most-negative-fixnum                    ; -2305843009213693952
;; Maximum number N of bits in safely-calculated integers.
integer-width                           ; 65536
```

3.2 Floating-Point Basics

The range of floating-point numbers is the same as the range of the C data type `double` on the machine you are using.

The read syntax for floating-point numbers requires either a decimal point, an exponent, or both. Optional signs (`+` or `-`) precede the number and its exponent.

```
;; 1500
1500.0
+15e2
15.0e+2
+1500000e-3
.15e4
```

Emacs Lisp requires at least one digit after a decimal point in a floating-point number that does not have an exponent. `1500.` is an integer.

```
;; read syntaxes for special floating-point values.
1.0e+INF                                ; 1.0e+INF
-1.0e+INF                               ; -1.0e+INF
0.0e+NaN                                ; 0.0e+NaN
-0.0e+NaN                               ; -0.0e+NaN
```

The following functions are specialized for handling floating-point numbers:

- `(isnan x)`
  This predicate returns `t` if its floating-point argument is a `NaN`, `nil` otherwise.

- `(frexp x)`
  This function returns a cons cell `(s . e)`, where `s` and `e` are respectively the significand and exponent of the floating-point number `x`.

- `(ldexp s e)`
  Given a numeric significand `s` and an integer exponent `e`, this function returns the floating point number $s2^e$.

- `(copysign x1 x2)`
  This function copies the sign of `x2` to the value of `x1`, and returns the result. `x1` and `x2` must be floating point.

- `(logb x)`
  This function returns the binary exponent of `x`.

3.3 Type Predicates for Numbers

- bignump
- fixnump
- floatp
- integerp
- numberp
- natnump
- zerop

natnum stands for natural number.

3.4 Comparison of Numbers

- `(= number-or-marker &rest number-or-markers)`
  This function tests whether all its arguments are numerically equal, and returns `t` if so, `nil` otherwise.
- `(eql value1 value2)`
  This function acts like `eq` except when both arguments are numbers. It compares numbers by type and numeric value. Floating-point values with the same sign, exponent and fraction are `eql`. This differs from numeric comparison: `(eql 0.0 -0.0)` returns `nil` and `(eql 0.0e+NaN 0.0e+NaN)` returns `t`, whereas `=` does the opposite.
- `(/= number-or-marker1 number-or-marker2)`
  This function tests whether its arguments are numerically equal, and returns `t` if they are not, and `nil` if they are.
- `(< number-or-marker &rest number-or-markers)`
  This function tests whether each argument is strictly less than the following argument. It returns `t` if so, `nil` otherwise.
- `(<= number-or-marker &rest number-or-markers)`
  This function tests whether each argument is less than or equal to the following argument. It returns `t` if so, `nil` otherwise.
- `(> number-or-marker &rest number-or-markers)`
  This function tests whether each argument is strictly greater than the following argument. It returns `t` if so, `nil` otherwise.
- `>= number-or-marker &rest number-or-markers`
  This function tests whether each argument is greater than or equal to the following argument. It returns `t` if so, `nil` otherwise.
- `(max number-or-marker &rest numbers-or-markers)`
  This function returns the largest of its arguments.

- `min number-or-marker &rest numbers-or-markers`
  This function returns the smallest of its arguments.

- `(abs number)`
  This function returns the absolute value of number.

3.5 Numeric Conversions

- `(float number)`
  This returns `number` converted to floating point. If `number` is already floating point, `float` returns it unchanged.

- `(truncate number &optional divisor)`
  This returns `number`, converted to an integer by rounding towards zero.

- `(floor number &optional divisor)`
  This returns number, converted to an integer by rounding downward (towards negative infinity).

- `(ceiling number &optional divisor)`
  This returns `number`, converted to an integer by rounding upward (towards positive infinity).

- `(round number &optional divisor)`
  This returns `number`, converted to an integer by rounding towards the nearest integer.

3.6 Arithmetic Operations

- `(1+ number-or-marker)`
  This function returns number-or-marker plus 1.

- `(1- numbers-or-markers)`
  This function returns number-or-marker minus 1.

- `(+ &rest numbers-or-markers)`
  This function adds its arguments together. When given no arguments, + returns 0.

- `(- &optional number-or-marker &rest more-numbers-or-markers)`
  The `-` function serves two purposes: negation and subtraction. When `-` has a single argument, the value is the negative of the argument. When there are multiple arguments, `-` subtracts each of the `more-numbers-or-markers` from `number-or-marker`, cumulatively. If there are no arguments, the result is 0.

- `(* &rest numbers-or-markers)`
  This function multiplies its arguments together, and returns the product. When given no arguments, `*` returns 1.

- `(/ number &rest divisors)`
  With one or more `divisors`, this function divides `number` by each divisor in `divisors` in turn, and returns the quotient. With no `divisors`, this function returns 1/`number`. Each argument may be a number or a marker. If all the arguments are integers, the result is an integer, obtained by rounding the quotient towards zero after each division.

- `(% dividend divisor)`
  This function returns the integer remainder after division of `dividend` by `divisor`. The arguments must be integers or markers.

- `(mod dividend divisor)`
  This function returns the value of `dividend` modulo `divisor`; in other words, the remainder after division of `dividend` by `divisor`, but with the same sign as `divisor`. The arguments must be numbers or markers.
  Unlike `%`, `mod` permits floating-point arguments; it rounds the quotient downward (towards minus infinity) to an integer, and uses that quotient to compute the remainder.

3.7 Rounding Operations

The functions `ffloor, fceiling, fround`, and `ftruncate` take a floating-point argument and return a floating-point result whose value is a nearby integer.

```
(floor 1.1)                                ; 1
(ffloor 1.1)                               ; 1.0
```

3.8 Bitwise Opeartions on Integers

The bitwise operations in Emacs Lisp apply only to integers.

- `(ash integer1 count)`
  `ash` (arithmetic shift) shifts the bits in `integer1` to the left `count` places, or to the right if `count` is negative. Left shifts introduce zero bits on the right; right shifts discard the rightmost bits.

- `(lsh integer1 count)`
  `lsh` (logical shift) shifts the bits in `integer1` to the left `count` places, or to the right if `count` is negative, bringing zeros into the vacated bits.

```
;; -7 = ...1111111111111111111111111111001
(ash -7 -1)                   ; -4 = ...1111111111111111111111111111100
(lsh -7 -1)           ; 536870908  = ...0111111111111111111111111111100
```

- `(logand &rest ints-or-markers)`
  This function returns the bitwise AND of the arguments: the nth bit is 1 in the result if, and only if, the nth bit is 1 in all the arguments. If `logand` is not passed any argument, it returns a value of -1. This number is an identity element for `logand` because its binary representation consists entirely of ones. If `logand` is passed just one argument, it returns that argument.

- `(logior &rest ints-or-markers)`
  This function returns the bitwise inclusive OR of its arguments: the nth bit is 1 in the result if, and only if, the nth bit is 1 in at least one of the arguments. If there are no arguments, the result is 0, which is an identity element for this operation. If `logior` is passed just one argument, it returns that argument.

- `(logxor &rest ints-or-markers)`
  This function returns the bitwise exclusive OR of its arguments: the nth bit is 1 in the result if, and only if, the nth bit is 1 in an odd number of the arguments. If there are no arguments, the result is 0, which is an identity element for this operation. If `logxor` is passed just one argument, it returns that argument.

- `(lognot integer)`
  This function returns the bitwise complement of its argument: the nth bit is one in the result if, and only if, the nth bit is zero in `integer`, and vice-versa.

- `(logcount integer)`
  This function returns the **Hamming weight** of `integer`: the number of ones in the binary representation of `integer`. If `integer` is negative, it returns the number of zero bits in its two's complement binary representation. The result is always nonnegative.

3.9 Standard Mathematical Functions

These mathematical functions allow integers as well as floating-point numbers as arguments.

- `(sin arg)`
- `(cos arg)`
- `(tan arg)`
- `(asin arg)`
- `(acos arg)`
- `(atan y &optional x)`
- `(exp arg)`
- `(log arg &optional base)`
- `(expt x y)`
- `(sqrt arg)`
- `float-e`
- `float-pi`

3.10 Random Numbers

A deterministic computer program cannot generate true random numbers. For most purposes, **pseudo-random numbers** suffice. A series of pseudo-random numbers is generated in a deterministic fashion. The numbers are not truly random, but they have certain properties that mimic a random series. For example, all possible values occur equally often in a pseudo-random series.

Pseudo-random numbers are generated from a **seed** value. Starting from any given seed, the random function always generates the same sequence of numbers. By default, Emacs initializes the random seed at startup, in such a way that the sequence of values of random (with overwhelming likelihood) differs in each Emacs run.

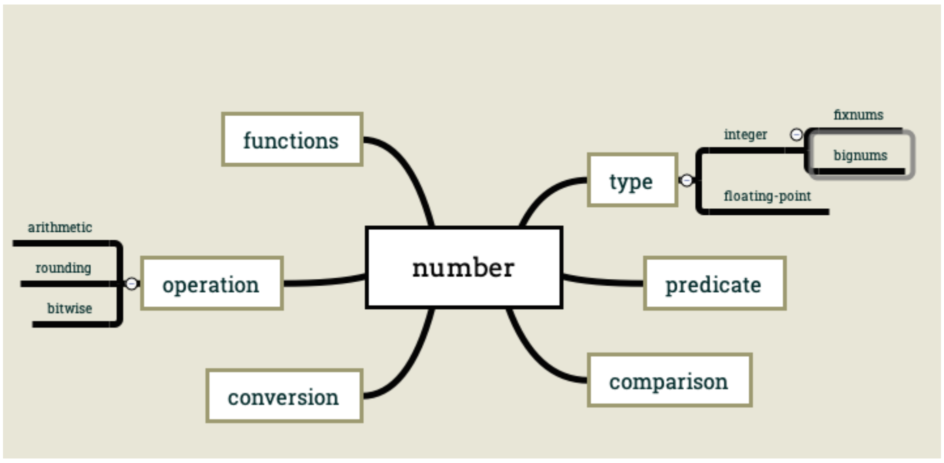
Sometimes you want the random number sequence to be repeatable. For example, when debugging a program whose behavior depends on the random number sequence, it is helpful to get the same behavior in each program run. To make the sequence repeat, execute `(random "")`. This sets the seed to a constant value for your particular Emacs executable (though it may differ for other Emacs builds). You can use other strings to choose various seed values.

```
(random &optional limit)
```

This function returns a pseudo-random integer. Repeated calls return a series of pseudo-random integers.

If `limit` is a positive integer, the value is chosen to be nonnegative and less than `limit`. Otherwise, the value might be any fixnum. If `limit` is `t`, it means to choose a new seed as if Emacs were restarting, typically from the system entropy. On systems lacking entropy pools, choose the seed from less-random volatile data such as the current time. If `limit` is a string, it means to choose a new seed based on the string's contents.

3.11 Summary



Figures 3.1: Numbers

Chapter 4

Strings and Characters

A string is an array that contains an ordered sequence of characters.

4.1 String and Character Basics

A character is a Lisp object which represents a single character of text. In Emacs Lisp, characters are simply integers; whether an integer is a character or not is determined only by how it is used.

A string is a fixed sequence of characters. It is a type of sequence called a **array**, meaning that its length is fixed and cannot be altered once it is created

There are two text representations for non-ASCII characters in Emacs strings (and in buffers): unibyte and multibyte.

Strings cannot hold characters that have the hyper, super or alt modifiers; they can hold ASCII control characters, but no other control characters. They do not distinguish case in ASCII control characters.

4.2 Predicates for Strings

♥ `(stringp object)`
This function returns `t` if `object` is a string, `nil` otherwise.

♥ `(string-or-null-p object)`
This function returns `t` if `object` is a string or `nil`. It returns `nil` otherwise

♥ `(char-or-string-p object)`
This function returns `t` if `object` is a string or a character (i.e., an integer), `nil` otherwise.

4.3 Creating Strings

♥ `(make-string count character &optional multibyte)`
This function returns a string made up of `count` repetitions of `character`. If `count` is negative, an error is signaled.

```
(make-string 5 ?x)                          ; "xxxxx"
```

Normally, if `character` is an ASCII character, the result is a unibyte string. But if the optional argument `multibyte` is non-nil, the function will produce a multibyte string instead.

♥ `(string &rest characters)`

This returns a string containing the characters `characters`.

```
(string ?a ?b ?c)                        ; "abc"
```

♥ `(substring string &optional start end)`

This function returns a new string which consists of those characters from `string` in the range from (and including) the character at the index `start` up to (but excluding) the character at the index `end`. The first character is at index zero.

If the characters copied from `string` have text properties, the properties are copied into the new string also

```
(substring "abcdefg" 0 3)                ; "abc"
(substring "abcdefg" -3 -1)              ; "ef"
(substring "abcdefg" -3 nil)             ; "efg"
(substring "abcdefg" 0)                  ; "abcdefg"
```

♥ `(substring-no-properties string &optional start end`

This works like `substring` but discards all text properties from the value.

♥ `(concat &rest sequences)`

This function returns a string consisting of the characters in the arguments passed to it (along with their text properties, if any). The arguments may be strings, lists of numbers, or vectors of numbers; they are not themselves changed. If `concat` receives no arguments, it returns an empty string.

```
(concat "abc" "-def")                    ; "abc-def"
(concat "abc" (list 120 121) [122])      ; "abcxyz"
(concat "abc" nil "-def")                ; "abc-def"
(concat)                                 ; ""
```

♥ `(split-string string &optional separators omit-nulls trim)`

This function splits `argument` into substrings based on the regular expression `separators`. Each match for `separators` defines a splitting point; the substrings between splitting points are made into a slit, which is returned.

If `separators` is `nil` (or omitted), the default is the value of `split-string-default-separators` and the function behaves as if `omit-nulls` were `t`.

If `omit-nulls` is `nil` (or omitted), the result contains null strings whenever there are two consecutive matches for separators, or a match is adjacent to the beginning or end of string. If `omit-nulls` is `t`, these null strings are omitted from the result.

If the optional argument `trim` is non-`nil`, it should be a regular expression to match text to trim from the beginning and end of each substring. If trimming makes the substring empty, it is treated as null.

```
(split-string " two words ")             ; ("two" "words")
(split-string "  two words "
              split-string-default-separators) ; ("" "two" "words" "")
(split-string "Soup is good food" "o")         ; ("S" "up is g" "" "d f" "" "
    d")
(split-string "Soup is good food" "o" t)       ; ("S" "up is g" "d f" "d")
(split-string "Soup is good food" "o+")        ; ("S" "up is g" "d f" "d")
```

- ♥ `split-string-default-separators`
  The default value of separators for `split-string`.

- ♥ `(string-clean-whitespace string)`
  Clean up the whitespace in `string` by collapsing stretches of whitespace to a single space character, as well as removing all whitespace from the start and the end of `string`. `(string-trim-left string &optional regexp)`
  Remove the leading text that matches `regexp` from `string`.

- ♥ `(string-trim-right string &optional regexp)`

- ♥ `(string-trim string &optional trim-left trim-right)`
  Remove the leading text that matches `trim-left` and trailing text that matches `trim-right` from string.

- ♥ `(string-fill string length)`
  Attempt to Word-wrap `string` so that no lines are longer than `length`. Filling is done on whitespace boundaries only.

- ♥ `(string-limit string length &optional end coding-system)`
  If `string` is shorter than `length` characters, `string` is returned as is. Otherwise, return a substring of `string` consisting of the first `length` characters. If the optional `end` parameter is given, return a string of the `length` last characters instead.
  If `coding-system` is non-`nil`, `string` will be encoded before limiting.

- ♥ `(string-lines string &optional omit-nulls)`
  Split `string` into a list of strings on newline boundaries. If `omit-nulls`, remove empty lines from the results.

- ♥ `(string-pad string length &optional padding start)`
  Pad `string` to be of the given `length` using `padding` as the padding character. `padding` defaults to the space character. If `string` is longer than `length`, no padding is done. If `start` is `nil` or omitted, the padding is appended to the characters of `string`, and if it's non-`nil`, the padding is prepended to `string`'s characters. `(string-chop-newline string)`
  Remove the final newline, if any, from `string`.

4.4 Modifying Strings

- ♥ `(aset string idx char)`
  This function stores `char` into `string` at character index `idx`.

- ♥ `(store-substring string idx obj)`
  This function alters part of the contents of the specified `string`, by storing `obj` starting at character index `idx`.

- ♥ `(clear-string string)`
  This makes `string` a unibyte string and clears its contents to zeros. It may also change `string`'s length.

4.5 Comparsion of Characters and Strings

♥ `(char-equal character1 character2)`
This function returns `t` if the arguments represent the same character, `nil` otherwise. This function ignores differences in case if `case-fold-search` is non-`nil`.

♥ `(string= string1 string2)`
This function returns `t` if the characters of the two strings match exactly. Symbols are also allowed as arguments, in which case the symbol names are used. Case is always significant, regardless of `case-fold-search`.

♥ `(string-equal string1 string2)`
another name for `string=`

♥ `(string-collate-equalp string1 string2 &optional locale ignore-case)`
This function returns `t` if `string1` and `string2` are equal with respect to collation rules. A **collation rule** is not only determined by the lexicographic order of the characters contained in `string1` and `string2`, but also further rules about relations between these characters. Usually, it is defined by the `locale` environment Emacs is running with and by the Standard C library against which Emacs was linked.
The optional argument `locale`, a string, overrides the setting of your current locale identifier for collation. The value is system dependent.
If `ignore-case` is non-`nil`, characters are converted to lower-case before comparing them.

♥ `(string< string1 string2)`
This function compares two strings a character at a time. It scans both the strings at the same time to find the first pair of corresponding characters that do not match. If the lesser character of these two is the character from `string1`, then `string1` is less, and this function returns `t`. If the lesser character is the one from `string2`, then `string1` is greater, and this function returns `nil`. If the two strings match entirely, the value is `nil`.

♥ `(string-lessp string1 string2)`
another name for `string<`

♥ `(string-greaterp string1 string2)`
This function returns the result of comparing `string1` and `string2` in the opposite order, i.e., it is equivalent to calling `(string-lessp string2 string1)`.

♥ `(string-collate-lessp string1 string2 &optional locale ignore-case)`
This function returns `t` if `string1` is less than `string2` in collation order.

♥ `(string-version-lessp string1 string2)`
This function compares strings lexicographically, except it treats sequences of numerical characters as if they comprised a base-ten number, and then compares the numbers. So ‘foo2.png’ is “smaller” than ‘foo12.png’ according to this predicate, even if ‘12’ is lexicographically “smaller” than ‘2’.

♥ `(string-prefix-p prefix string &optional ignore-case)`
This function returns non-`nil` if `prefix` is a prefix of `string`.

♥ `(string-suffix-p suffix string &optional ignore-case)`
This function returns non-`nil` if `suffix` is a suffix of `string`.

- ♥ `(compare-strings string1 start1 end1 string2 start2 end2 &optional ignore-case)`
  The strings are compared by the numeric values of their characters.
  If the specified portions of the two strings match, the value is `t`. Otherwise, the value is an integer which indicates how many leading characters agree, and which string is less. Its absolute value is one plus the number of characters that agree at the beginning of the two strings. The sign is negative if `string1` (or its specified portion) is less.

- ♥ `(string-distance string1 string2 &optional bytecompare)`
  This function returns the **Levenshtein distance** between the source string `string1` and the target string `string2`. The Levenshtein distance is the number of single character changes — deletions, insertions, or replacements — required to transform the source string into the target string; it is one possible definition of the edit distance between strings.

4.6 Conversion of Characters and Strings

This section describes functions for converting between characters, strings and integers.

- ♥ `(number-to-string number)`
  This function returns a string consisting of the printed base-ten representation of number.

- ♥ `(string-to-number string &optional base)`
  This function returns the numeric value of the characters in string. If `string` cannot be interpreted as a number, this function returns 0.

4.7 Formatting Strings

Formatting means constructing a string by substituting computed values at various places in a constant string. This constant string controls how the other values are printed, as well as where they appear; it is called a **format string**.

- ♥ `(format string &rest objects)`
  This function returns a string equal to `string`, replacing any format specification with encodings of the corresponding `objects`. The argument `objects` are the computed values to be formatted.
  The characters in `string`, other than the format specifications, are copied directly into the output, including their text properties, if any. Any text properties of the format specifications are copied to the produced string representations of the argument `objects`.

```
(format "hello, %s" "Ming")              ; "hello, Ming"
```

- ♥ `(format-message string &rest objects)`
  This function acts like `format`, except it also converts any grave accents (‘) and apostrophes (’) in string as per the value of `text-quoting-style`.

```
(format "hello, '%s'" "Ming")            ; "hello, 'Ming'"
(format-message "hello, '%s'" "Ming")    ; "hello, 'Ming’"
```

A **format specification** is a sequence of characters beginning with a `%`. Certain format specifications require values of particular types. If you supply a value that doesn’t fit the requirements, an error is signaled.

Here is a list of valid format specifications:

- %s
  Replace the specification with the printed representation of the object, made without quoting.
- %S
  Replace the specification with the printed representation of the object, made with quoting.

```
(format "%s" "hello")                    ; "hello"
(format "%S" "hello")                    ; "\"hello\""
```

- %o
  Replace the specification with the base-eight representation of an integer.
- %d
  Replace the specification with the base-ten representation of a signed integer.
- %x
  Replace the specification with the base-sixteen representation of an integer using lower case.
- %X
  Replace the specification with the base-sixteen representation of an integer using upper case.
- %c
  Replace the specification with the character which is the value given.
- %e
  Replace the specification with the exponential notation for a floating-point number.
- %f
  Replace the specification with the decimal-point notation for a floating-point number.
- %g
  Replace the specification with notation for a floating-point number, using either exponential notation or decimal-point notation.
- %%
  Replace the specification with a single %. This format specification is unusual in that its only form is plain %% and that it does not use a value.

```
(format "%% %d" 30)                      ; "% 30"
```

By default, format specifications correspond to successive values from `objects`. Thus, the first format specification in string uses the first such value, the second format specification uses the second such value, and so on. Any extra format specifications (those for which there are no corresponding values) cause an error. Any extra values to be formatted are ignored.

A format specification can have a **field number**, which is a decimal number immediately after the initial %, followed by a literal dollar sign $. It causes the format specification to convert the argument with the given number instead of the next argument. Field numbers start at 1. A format can contain either numbered or unnumbered format specifications but not both, except that %% can be mixed with numbered specifications.

```
(format "%2$s, %3$s, %%, %1$s" "x" "y" "z")
;;  "y, z, %, x"
```

After the % and any field number, you can put certain **flag characters**.

- +
  Insert a plus sign before a nonnegative number. They are ignored except for %d, %e, %f, %g.

- space character
  Insert a space before a nonnegative number. They are ignored except for %d, %e, %f, %g. If both + and space character are used, + takes precedence.

- #
  Specifies an alternate form which depends on the format in use.
  - For %o, it ensures that the result begins with a 0.
  - For %x and %X, it prefixes nonzero results with 0x or 0X.
  - For %e and %f, it include a decimal point even if the precision is zero.
  - For %g, it always includes a decimal point, and also forces any trailing zeros after the decimal point to be left in place where they would otherwise be removed.

- 0
  Ensures that the padding consists of 0 characters instead of spaces. This flag is ignored for non-numerical specification characters like %s, %S and %c. These specification characters accept the 0 flag, but still pad with spaces.

- -
  Causes any padding inserted by the width, if specified, to be inserted on the right rather than the left. If both - and 0 are present, the 0 flag is ignored.

```
(format "%06d is padded on the left with zeros" 123)
;;  "000123 is padded on the left with zeros"
(format "'%-6d' is padded on the right" 123)
;;  "'123   ' is padded on the right"
(format "The word '%-7s' actually has %d letters in it."
        "foo" (length "foo"))
;;  "The word 'foo    ' actually has 3 letters in it."
```

A specification can have a **width**, which is a decimal number that appears after any field number and flags. If the printed representation of the object contains fewer characters than this width, format extends it with padding. Any padding introduced by the width normally consists of spaces inserted on the left. If the width is too small, format does not truncate the object's printed representation.

```
(format "%5d is padded on the left with spaces" 123)
;;  "  123 is padded on the left with spaces"
(format "The word '%7s' has %d letters in it."
        "specification" (length "specification"))
;;  "The word 'specification' has 13 letters in it."
```

All the specification characters allow an optional **precision** after the field number, flags and width, if present. The precision is a decimal-point . followed by a digit-string.

- For the floating-point specifications (%e and %f), the precision specifies how many digits following the decimal point to show; if zero, the decimal-point itself is also omitted.

- For %g, the precision specifies how many significant digits to show (significant digits are the first digit before the decimal point and all the digits after it). If the precision of %g is zero or unspecified, it is treated as 1.

- For %s and %S, the precision truncates the string to the given width.
- For other specification characters, the effect of precision is what the local library functions of the printf family produce.

4.8 Custom Format Strings

```
(format-spec template spec-alist &optional ignore-missing split)
```

This function returns a string produced from the format string template according to conversion specified in spec-alist, which is an alist of the form (letter . replacement). Each specification %letter in template will be replaced by replacement when formatting the resulting string.

Using an alist to specify conversions gives rise to some useful properties:

- If spec-alist contains more unique letter keys than there are unique specification characters in template, the unused keys are ignored.
- If spec-alist contains more than one association with the same letter, the closest on to the start of the list is used.
- If template contains the same specification character more than once, then the same replacement found in spec-alist is used as a basis for all of that character's substitutions.
- The order of specification in template need not to correspond to the order of association in spec-alist.

The optional argument ignore-missing indicates how to handle specification characters in template that are not found in spec-alist. If it is nil or omitted, the function signals an error; if it is ignore, those format specifications are left verbatim in the output, including their text properties, if any; if it is delete, those format specifications are removed from the output; any other non-nil value is handled like ignore, but any occurrences of %% are also left verbatim in the output.

If the optional argument split is non-nil, instead of returning a single string, format-spec will split the result into a list of strings, based on where the substitutions were performed.

```
(format-spec "foo %b bar" '((?b . "zot")) nil t)
;; ("foo " "zot" " bar")
```

Unlike format, which assigns specific meanings to a fixed set of specification characters, format-spec accepts arbitrary specification characters and treats them all equally. For example:

```

(setq my-site-info
      (list (cons ?s system-name)
            (cons ?t (symbol-name system-type))
            (cons ?c system-configuration)
            (cons ?v emacs-version)
            (cons ?e invocation-name)
            (cons ?p (number-to-string (emacs-pid)))
            (cons ?a user-mail-address)
            (cons ?n user-full-name)))

(format-spec "%e %v (%c) Emacs User: %n" my-site-info)
;; "Emacs-x86_64-10_14 28.2 (x86_64-apple-darwin18.7.0) Emacs User: Mingming Li"
```

A format specification can include any number of the following flag characters immedi- ately after the `%`.

- `0`
  Causes any padding specified by the width to consist of `0` characters instead of spaces.
- `-`
  Causes any padding specified by the width to be inserted on the right rather than the left.
- `<`
  Causes the substitution to be truncated on the left to the given width and precision, if specified.
- `>`
  Causes the substitution to be truncated on the right to the given width, if specified.
- `^`
  Converts the substituted text to upper case.
- `^_`
  Converts the substituted text to lower case.

As is the case with `format`, a format specification can include a width, which is a decimal number that appears after any flags, and a precision, which is a decimal-point `.` followed by a decimal number that appears after any flags and width.

If a substitution contains fewer characters than its specified width, it is padded on the left. If a substitution contains more characters than its specified precision, it is truncated on the right.

```
(setq my-battery-info
      (list (cons ?p "73")      ; Percentage
            (cons ?L "Battery") ; Status
            (cons ?t "2:23")    ; Remaining time
            (cons ?c "24330")   ; Capacity
            (cons ?r "10.6")))  ; Rate of discharge

(format-spec "%>^-3L : %3p%% (%05t left)" my-battery-info)
;; "BAT :  73% (02:23 left)"
```

4.9 Case Conversion in Lisp

The character case functions change the case of single characters or of the contents of strings. The functions normally convert only alphabetic characters (the letters ‘A’ through ‘Z’ and ‘a’ through ‘z’, as well as non-ASCII letters); other characters are not altered. You can specify a different case conversion mapping by specifying a case table.

These functions do not modify the strings that are passed to them as arguments.

- (downcase string-or-char)
- (upcase string-or-char)
- (capitalize string-or-char)
- (upcase-initials string-or-char)

```
;; downcase
(downcase "The cat in the hat")          ; "the cat in the hat"
(downcase ?X)                            ; 120

;; upcase
(upcase "The cat in the hat")            ; "THE CAT IN THE HAT"
(upcase ?x)                              ; 88

;; capitalize
(capitalize "The cAt in the hat")        ; "The Cat In The Hat"
(capitalize ?x)                          ; 88

;; upcase-initials
(upcase-initials "The cAt in the hat")  ; "The CAt In The Hat"
(upcase-initials ?x)                     ; 88
```

4.10 The Case Table

You can customize case conversion by installing a special **case table**. A case table specifies the mapping between upper case and lower case letters. It affects both the case conversion functions for Lisp objects and those that apply to text in the buffer. Each buffer has a case table; there is also a standard case table which is used to initialize the case table of new buffers.

A case table is a char-table whose subtype is `case-table`. This char-table maps each character into the corresponding lower case character. It has three extra slots, which hold related tables:

- `upcase`
  The upcase table maps each character into the corresponding upper case character.
- `canonicalize`
  The canonicalize table maps all of a set of case-related characters into a particular member of that set.
- `equivalences`
  The equivalences table maps each one of a set of case-related characters into the next character in that set.

In simple cases, all you need to specify is the mapping to lower-case; the three related tables will be calculated automatically from that one.

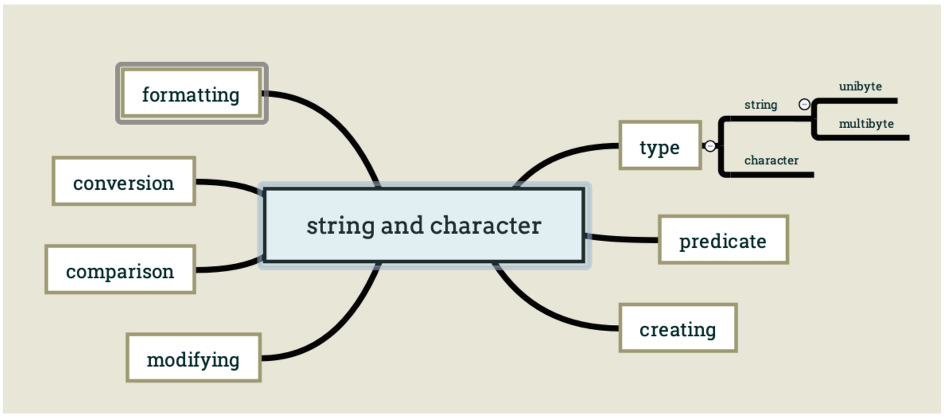
For some languages, upper and lower case letters are not in one-to-one correspondence. There may be two different lower case letters with the same upper case equivalent. In these cases, you need to specify the maps for both lower case and upper case.

The extra table `canonicalize` maps each character to a canonical equivalent; any two characters that are related by case-conversion have the same canonical equivalent character. For example, since 'a' and 'A' are related by case-conversion, they should have the same canonical equivalent character (which should be either 'a' for both of them, or 'A' for both of them).

The extra table `equivalences` is a map that cyclically permutes each equivalence class (of characters with the same canonical equivalent). (For ordinary ASCII, this would map 'a' into 'A' and 'A' into 'a', and likewise for each set of equivalent characters.)

When constructing a case table, you can provide `nil` for `canonicalize`; then Emacs fills in this slot from the lower case and upper case mappings. You can also provide `nil` for `equivalences`; then Emacs fills in this slot from `canonicalize`. In a case table that is actually in use, those components are non-`nil`.

4.11 Summary



Figures 4.1: String and Character

Chapter 5

Lists

A **list** represents a sequence of zero or more elements (which may be any Lisp objects).

5.1 Lists and Cons Cells

A list is a series of **cons cells** chained together, so that each cell refers to the next one. There is one cons cell for each element of the list. By convention, the cars of the cons cells hold the elements of the list, and the cdrs are used to chain the list (this asymmetry between car and cdr is entirely a matter of convention; at the level of cons cells, the car and cdr slots have similar properties).

Also by convention, the cdr of the last cons cell in a list is `nil`. We call such a `nil`-terminated structure a **proper list**. If the cdr of a list's last cons cell is some value other than `nil`, we call the structure a **dotted list**, since its printed representation would use dotted pair notation. There is one other possibility: some cons cell's cdr could point to one of the previous cons cells in the list. We call that structure a **circular list**.

Because most cons cells are used as part of lists, we refer to any structure made out of cons cells as a **list structure**.

5.2 Predicates on Lists

- `(consp object)`
  This function returns `t` if object is a cons cell, `nil` otherwise. `nil` is not a cons cell, although it is a list.

- `(atom object)`
  This function returns `t` if object is an atom, `nil` otherwise. All objects except cons cells are atoms. The symbol `nil` is an atom and is also a list; it is the only Lisp object that is both.

- `(listp object)`
  This function returns `t` if object is a cons cell or `nil`. Otherwise, it returns `nil`.

- `(nlistp object)`
  This function is the opposite of listp: it returns `t` if object is not a list. Otherwise, it returns `nil`.

- `(null object)`
  This function returns `t` if object is `nil`, and returns `nil` otherwise.

- ♥ `(proper-list-p object)`
  This function returns the length of object if it is a proper list, `nil` otherwise.

5.3 Accessing Elements of Lists

- ♥ `(car cons-cell)`
  This function returns the value referred to by the first slot of the cons cell `cons-cell`. If `cons-cell` is `nil`, this function returns `nil`. An error is signaled if the argument is not a cons cell or `nil`.

- ♥ `(cdr cons-cell)`
  This function returns the value referred to by the second slot of the cons cell `cons-cell`. If `cons-cell` is `nil`, this function returns `nil`. An error is signaled if the argument is not a cons cell or `nil`.

- ♥ `(car-safe object)`
  This function lets you take the car of a cons cell while avoiding errors for other data types. It returns the car of `object` if `object` is a cons cell, `nil` otherwise.

- ♥ `(cdr-safe object)`
  This function lets you take the cdr of a cons cell while avoiding errors for other data types. It returns the cdr of `object` if `object` is a cons cell, `nil` otherwise.

- ♥ `(pop listname)`
  This macro provides a convenient way to examine the car of a list, and take it off the list, all at once. It operates on the list stored in `listname`. It removes the first element from the list, saves the cdr into `listname`, then returns the removed element.

- ♥ `(nth n list)`
  This function returns the nth element of list. If the length of list is `n` or less, the value is `nil`.

- ♥ `(nthcdr n list)`
  This function returns the nth cdr of list.

- ♥ `(last list &optional n)`
  This function returns the last link of list. The car of this link is the list's last element. If `list` is null, `nil` is returned. If `n` is non-`nil`, the nth-to-last link is returned instead, or the whole of list if `n` is bigger than list's length.

- ♥ `(safe-length list)`
  This function returns the length of list, with no risk of either an error or an infinite loop. If `list` is not `nil` or a cons cell, safe-length returns 0.

- ♥ `(butlast x &optional n)`
  This function returns the list `x` with the last element, or the last `n` elements, removed. If `n` is greater than zero it makes a copy of the list so as not to damage the original list. In general, `(append (butlast x n)( last x n))` will return a list equal to `x`.

- ♥ `(nbutlast x &optional n)`
  This is a version of `butlast` that works by destructively modifying the cdr of the appropriate element, rather than making a copy of the list.

5.4 Building Cons Cells and Lists

♥ `(cons object1 object2)`

This function is the most basic function for building new list structure. It creates a new cons cell, making `object1` the car, and `object2` the cdr. It then returns the new cons cell.

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

♥ `(list &rest objects)`

This function creates a list with `objects` as its elements. The resulting list is always `nil`-terminated. If no `objects` are given, the empty list is returned.

```
(list 1 2 3 4 5)                        ; (1 2 3 4 5)
(list 1 2 '(3 4 5) 'foo)                ; (1 2 (3 4 5) foo)
(list)                                  ; nil
```

♥ `(make-list length object)`

This function creates a list of `length` elements, in which each element is `object`.

```
(make-list 3 'pigs)                     ; (pigs pigs pigs)
```

♥ `(append &rest sequences)`

This function returns a list containing all the elements of sequences. All arguments except the last one are copied, so none of the arguments is altered. The final argument is not copied or converted; it becomes the cdr of the last cons cell in the new list.

```
(setq trees '(pine oak))                ; (pine oak)
(setq more-trees (append '(maple birch) trees)) ; (maple birch pine oak)
trees                                           ; (pine oak)
more-trees                                      ; (maple birch pine oak)
(append [a b] "cd" nil)                         ; (a b 99 100)
(append)                                        ; nil
(append '(x y) 'z)                              ; (x y . z)
```

♥ `(copy-tree tree &optional vecp)`

This function returns a copy of the tree `tree`. If `tree` is a cons cell, this makes a new cons cell with the same car and cdr, then recursively copies the car and cdr in the same way.

Normally, when `tree` is anything other than a cons cell, `copy-tree` simply returns tree. However, if `vecp` is non-nil, it copies vectors too (and operates recursively on their elements).

♥ `(flatten-tree tree)`

This function returns a “flattened” copy of tree, that is, a list containing all the non-`nil` terminal nodes, or leaves, of the tree of cons cells rooted at `tree`.

```
(flatten-tree '(1 (2 . 3) nil (4 5 (6)) 7)) ; (1 2 3 4 5 6 7)
```

♥ `(ensure-list object)`

This function returns `object` as a list. If `object` is already a list, the function returns it; otherwise, the function returns a one-element list containing `object`.

♥ `(number-sequence from &optional to separation)`
This function returns a list of numbers starting with `from` and incrementing by `separation`, and ending at or just before `to`.

```
(number-sequence 1.5 6 2)                ; (1.5 3.5 5.5)
```

5.5 Modifying List Variables

♥ `(push element listname)`
This macro creates a new list whose car is `element` and whose cdr is the list specified by `listname`, and saves that list in `listname`.

```
(setq l '(a b))                          ; (a b)
(push 'c l)                              ; (c a b)
l                                        ; (c a b)
```

♥ `(pop listname)`
This macro provides a convenient way to examine the car of a list, and take it off the list, all at once. It removes the first element from the list, saves the cdr into `listname`, then returns the removed element.

♥ `(add-to-list symbole element &optional append compare-fn)`
This function sets the variable `symbol` by consing `element` onto the old value, if `element` is not already a member of that value. It returns the resulting list, whether updated or not.

```
(setq foo '(a b))                        ; (a b)
(add-to-list 'foo 'c)                    ; (c a b)
(add-to-list 'foo 'b)                    ; (c a b)
foo                                      ; (c a b)
(add-to-list 'foo 'd t)                  ; (c a b d)
```

♥ `(add-to-ordered-list symbol element &optional order)`
This function sets the variable `symbol` by inserting `element` into the old value, which must be a list, at the position specified by `order`. If `element` is already a member of the list, its position in the list is adjusted according to `order`. Membership is tested using `eq`. This function returns the resulting list, whether updated or not.

```
(setq foo '())                           ;  nil
(add-to-ordered-list 'foo 'a 1)          ; (a)
(add-to-ordered-list 'foo 'c 3)          ; (a c)
(add-to-ordered-list 'foo 'b 2)          ; (a b c)
(add-to-ordered-list 'foo 'b 4)          ; (a c b)
(add-to-ordered-list 'foo 'd)            ; (a c b d)
(add-to-ordered-list 'foo 'e)            ; (a c b e d)
foo                                      ; (a c b e d)
```

5.6 Modifying Existing List Structure

You can modify the car and cdr contents of a cons cell with the primitives `setcar` and `setcdr`. These are destructive operations because they change existing list structure.

♥ (setcar cons object)

This function stores object as the new car of cons, replacing its previous car. It returns the value object.

```
(setq x (list 1 2))                     ; (1 2)
(setcar x 4)                            ; 4
x                                       ; (4 2)

;; Create two lists that are partly shared.
(setq x1 (list 'a 'b 'c))               ; (a b c)
(setq x2 (cons 'z (cdr x1)))            ; (z b c)
;; Replace the car of a shared link.
(setcar (cdr x1) 'foo)                  ; foo
x1                                      ; (a foo c)
x2                                      ; (z foo c)
;; Replace the car of a link that is not shared.
(setcar x1 'baz)                        ; baz
x1                                      ; (baz foo c)
x2                                      ; (z foo c)
```

♥ (setcdr cons object)

This function stores object as the new cdr of cons, replacing its previous cdr. It returns the value object.

```
(setq x (list 1 2 3))                   ; (1 2 3)
(setcdr x '(4))                         ; (4)
x                                       ; (1 4)

(setq x1 (list 'a 'b 'c))               ;  (a b c)
(setcdr x1 (cdr (cdr x1)))              ; (c)
x1                                      ; (a c)

(setq x1 (list 'a 'b 'c))               ; (a b c)
(setcdr x1 (cons 'd (cdr x1)))          ; (d b c)
x1                                      ; (a d b c)
```

♥ (nconc &rest lists)

This function returns a list containing all the elements of lists. Unlike append, the lists are not copied. Instead, the last cdr of each of the lists is changed to refer to the following list. The last of the lists is not altered.

Since the last argument of nconc is not itself modified, it is reasonable to use a constant list. However, the other arguments (all but the last) should be mutable lists.

```
(setq x (list 1 2 3))                   ; (1 2 3)
(nconc x '(4 5))                        ; (1 2 3 4 5)
x                                       ; (1 2 3 4 5)
```

5.7 Using Lists as Sets

♥ (memq object list)

This function tests to see whether object is a member of list. If it is, memq returns a list starting with the first occurrence of object. Otherwise, it returns nil. The letter 'q' in memq says that it uses eq to compare object against the elements of the list.

```
(memq 'b '(a b c b a))                 ; (b c b a)
```

♥ (delq object list)
This function destructively removes all elements eq to object from list, and returns the resulting list. The delq function deletes elements from the front of the list by simply advancing down the list, and returning a sublist that starts after those elements. When an element to be deleted appears in the middle of the list, removing it involves changing the cdrs.

```
(delq 'a '(a b c))                     ; (b c)

(setq sample-list (list 'a 'b 'c '(4))) ; (a b c (4))
(delq 'a sample-list)                  ; (b c (4))
sample-list                            ; (a b c (4))
(delq 'c sample-list)                  ; (a b (4))
sample-list                            ; (a b (4))
```

♥ (remq object list)
This function returns a copy of list, with all elements removed which are eq to object.

```
(setq sample-list (list 'a 'b 'c 'a 'b 'c)) ; (a b c a b c)
(remq 'a sample-list)                       ; (b c b c)
sample-list                                 ; (a b c a b c)
```

♥ (memql object list)
This function tests to see whether object is a member of list, comparing members with object using eql. If object is a member, memql returns a list starting with its first occurrence in list. Otherwise, it returns nil.

```
(memql 1.2 '(1.1 1.2 1.3))             ; (1.2 1.3)
(memq 1.2 '(1.1 1.2 1.3))              ; nil
```

♥ (member object list)
This function tests to see whether object is a member of list, comparing members with object using equal. If object is a member, member returns a list starting with its first occurrence in list. Otherwise, it returns nil.

```
(member '(2) '((1) (2)))               ; ((2))
(memq '(2) '((1) (2)))                 ; nil
```

♥ (delete object sequence)
This function removes all elements equal to object from sequence, and returns the resulting sequence. If sequence is a list, it delete likes delq but comparing with equal. If sequence is a vector or string, delete returns a copy of sequence with all elements equal to object removed.

```
(setq l (list '(2) '(1) '(2)))         ; ((2) (1) (2))
(delete '(2) l)                        ; ((1))
l                                      ; ((2) (1))

(setq l [(2) (1) (2)])                 ; [(2) (1) (2)]
```

```
(delete '(2) l)                         ; [(1)]
l                                       ; [(2) (1) (2)]
```

♥ `(remove object sequence)`
This function is the non-destructive counterpart of `delete`. It returns a copy of `sequence`, a list, vector, or string, with elements `equal` to object removed.

♥ `(member-ignore-case object list)`
This function is like `member`, except that `object` should be a string and that it ignores differences in letter-case and text representation.

♥ `(delete-dups list)`
This function destructively removes all `equal` duplicates from `list`, stores the result in `list` and returns `it`. Of several `equal` occurrences of an element in `list`, `delete-dups` keeps the first one.

5.8 Association Lists

An **association list**, or **alist** for short, records a mapping from keys to values. It is a list of cons cells called **associations**: the car of each cons cell is the key, and the cdr is the associated value.

♥ `(assoc key alist &optional testfn)`
This function returns the first association for `key` in `alist`, comparing `key` against the alist elements using `testfn` if it is a function, and `equal` otherwise. The function returns `nil` if no association in `alist` has a car equal to `key`, as tested by `testfn`.

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
;; ((pine . cones) (oak . acorns) (maple . seeds))
(assoc 'oak trees)
;;  (oak . acorns)
(cdr (assoc 'oak trees))
;;  acorns
(assoc 'birch trees)
;;  nil
```

♥ `(assoc-string key alist &optional case-fold)`
This function works like `assoc`, except that `key` must be string or symbol, and comparison is done using `compare-strings`. Symbols are converted to strings before testing. If `case-fold` is non-`nil`, `key` and the elements of `alist` are converted to uppercase before comparison.

♥ `(rassoc value alist)`
This function returns the first association with value `value` in `alist`. It returns `nil` if no association in `alist` has a cdr `equal` to `value`.

♥ `(assq key alist)`
This function is like `assoc` in that it returns the first association for key in `alist`, but it makes the comparison using `eq`.

♥ `(alist-get key alist &optional default remove testfn)`
It finds the first association by comparing key with `alist` elements, and, if found, return the value of that association. If no association is found, the function returns `default`. Comparison of `key` against `alist` elements uses the function specified by `testfn`, defaulting to `eq`.

- ♥ `(rassq value alist)`
  This function returns the first association with value `value` in `alist`. It returns `nil` if no association in `alist` has a cdr `eq` to value.

- ♥ `(assoc-default key alist &optional testfn default)`
  This function searches `alist` for a match for `key`. For each element of `alist`, it compares the element (if it is an atom) or the element's car (if it is a cons) against `key`, by calling `testfn` with two arguments: the element or its car, and `key`.
  If an `alist` element matches `key` by this criterion, then `assoc-default` returns a value based on this element. If the element is a cons, then the value is the element's cdr. Otherwise, the return value is `default`.
  If no alist element matches key, `assoc-default` returns `nil`.

- ♥ `(copy-alist alist)`
  This function returns a two-level deep copy of `alist`.

- ♥ `(assq-delete-all key alist)`
  This function deletes from `alist` all the elements whose car is `eq` to `key`, much as if you used `delq` to delete each such element one by one. It returns the shortened alist, and often modifies the original list structure of `alist`. For correct results, use the return value of `assq-delete-all` rather than looking at the saved value of `alist`.

- ♥ `(assoc-delete-all key alist &optional testfn)`
  This function is like `assq-delete-all` except that it accepts an optional argument `testfn`. If omitted or `nil`, `testfn` defaults to `equal`.

- ♥ `(rassq-delete-all value alist)`
  This function deletes from `alist` all the elements whose cdr is `eq` to value. It returns the shortened alist, and often modifies the original list structure of alist. `rassq-delete-all` is like `assq-delete-all` except that it compares the cdr of each alist association instead of the car.

- ♥ `(let-alist alist body)`
  Creates a binding for each symbol used as keys, prefixed with dot. This can be useful when accessing several items in the same association list.

```
(setq colors '((rose . red) (lily . white) (buttercup . yellow)))
;;  ((rose . red) (lily . white) (buttercup . yellow))
(let-alist colors
  (if (eq .rose 'red)
      .lily))
;;  white
```

The `body` is inspected at compilation time, and only the symbols that appear in body with a '.' as the first character in the symbol name will be bound. Finding the keys is done with `assq`. Nested association lists is supported:

```
(setq colors '((rose . red) (lily (belladonna . yellow) (brindisi . pink))))
;;  ((rose . red) (lily (belladonna . yellow) (brindisi . pink)))
(let-alist colors
  (if (eq .rose 'red)
      .lily.belladonna))
;;  yellow
```

5.9 Property Lists

A **property list** (**plist** for short) is a list of paired elements. Each of the pairs associates a property name (usually a symbol) with a property or value.

```
(pine cones numbers (1 2 3) color "blue")
```

This property list associates `pine` with `cones`, `numbers` with `(1 2 3)`, and `color` with `"blue"`.

♥ `(plist-get plist property)`
This returns the value of the `property` property stored in the property list `plist`. If `property` is not found in the `plist`, it returns `nil`.

```
(plist-get '(foo 4) 'foo)              ; 4
(plist-get '(foo 4 bad) 'foo)          ; 4
(plist-get '(foo 4 bad) 'bad)          ; nil
```

♥ `(plist-put plist property value)`
This stores `value` as the value of the `property` property in the property list `plist`. The function returns the modified property list.

```
(setq my-plist (list 'bar t 'foo 4))     ; (bar t foo 4)
(setq my-plist (plist-put my-plist 'foo 69)) ; (bar t foo 69)
(setq my-plist (plist-put my-plist 'quux '(a))) ; (bar t foo 69 quux (a))
```

♥ `(lax-plist-get plist property)`
Like `plist-get` except that it compares properties using `equal` instead of `eq`.

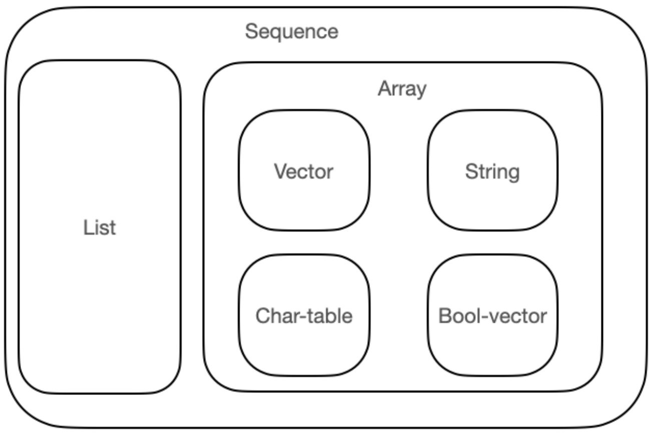
♥ `(lax-plist-put plist property value)`
Like `plist-put` except that it compares properties using `equal` instead of `eq`.

♥ `(plist-member plist property)`
This returns non-`nil` if `plist` contains the given `property`. Unlike `plist-get`, this allows you to distinguish between a missing property and a property with the value `nil`.

```
(plist-get '(foo 4 bad) 'bad)          ; nil
(plist-get '(foo 4 bad) 'bar)          ; nil
(plist-member '(foo 4 bad) 'bad)       ; (bad)
(plist-member '(foo 4 bad) 'bar)       ; nil
```

Chapter 6

Sequences, Arrays, and Vectors



Figures 6.1: Sequences

6.1 Sequences

♥ `(sequencep object)`
This function returns `t` if `object` is a sequence, `nil` otherwise.

♥ `(length sequence)`
This function returns the number of elements in sequence.

```
(length '(1 2 3))                     ; 3
```

♥ `(length< sequence length)`
Return non-`nil` if `sequence` is shorter than `length`. This may be more efficient than computing the length of sequence if `sequence` is a long list.

♥ `(length> sequence length)`

♥ `(length= sequence length)`

♥ `(elt sequence index)`
This function returns the element of sequence indexed by index.

```
(elt '(1 2 3 4) 2)                          ; 3
```

♥ `(copy-sequence seqr)`
This function returns a copy of `seqr`, which should be either a sequence or a record.

♥ `(reverse sequence)`
This function creates a new sequence whose elements are the elements of `sequence`, but in reverse order. The original argument `sequence` is not altered.

♥ `(nreverse sequence)`
This function reverses the order of the elements of `sequence`. Unlike `reverse` the original `sequence` may be modified.

♥ `(sort sequence predicate)`
This function sorts `sequence` stably. Note that this function doesn't work for all sequences; it may be used only for lists and vectors.

♥ `(seqp)`
This function returns non-nil if object is a sequence (a list or array), or any additional type of sequence defined via `seq.el` generic functions. This is an extensible variant of `sequencep`.

♥ `(seq-drop sequence n)`
This function returns all but the first `n` (an integer) elements of `sequence`. If `n` is negative or zero, the result is `sequence`.

```
(seq-drop [1 2 3 4 5 6] 3)                  ; [4 5 6]
```

♥ `(seq-take sequence n)`
This function returns the first `n` (an integer) elements of `sequence`. If `n` is negative or zero, the result is nil.

```
(seq-take '(1 2 3 4) 3)                     ; (1 2 3)
```

♥ `(seq-take-while predicate sequence)`
This function returns the members of `sequence` in order, stopping before the first one for which `predicate` returns nil.

```
(seq-take-while (lambda (elt) (> elt 0)) '(1 2 3 -1 -2))
; (1 2 3)
```

♥ `(seq-drop-while predicate sequence)`

♥ `(seq-do function sequence)`
This function applies function to each element of `sequence` in turn (presumably for side effects), and returns `sequence`.

♥ `(seq-map function sequence)`
This function returns the result of applying `function` to each element of `sequence`. The returned value is a list.

```
(seq-map #'1+ '(2 4 6))                    ; (3 5 7)
```

♥ `(seq-map-indexed function sequence)`
This function returns the result of applying `function` to each element of `sequence` and its index within seq. The returned value is a list.

```
(seq-map-indexed (lambda (elt idx)
                   (list idx elt))
                 '(a b c))
;;  ((0 a) (1 b) (2 c))
```

♥ `(seq-mapn function &rest sequences)`
This function returns the result of applying function to each element of sequences.

```
(seq-mapn #'+ '(2 4 6) '(20 40 60))
;;  (22 44 66)
```

♥ `(seq-filter predicate sequence)`
This function returns a list of all the elements in `sequence` for which `predicate` returns non-nil.

```
(seq-filter (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
;;  (1 3 5)
```

♥ `(seq-remove predicate sequence)`
This function returns a list of all the elements in `sequence` for which `predicate` returns nil.

```
(seq-remove (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
;;  (-1 -3)
```

♥ `(seq-reduce function sequence initial-value)`
This function returns the result of calling `function` with `initial-value` and the first element of `sequence`, then calling `function` with that result and the second element of `sequence`, then with that result and the third element of `sequence`, etc. `function` should be a function of two arguments.

```
(seq-reduce #'+ '(1 2 3 4) 5)
;;  15
```

♥ `(seq-some predicate sequence)`
This function returns the first non-nil value returned by applying `predicate` to each element of sequence in turn.

```
(seq-some #'numberp ["abc" 1 nil])       ; t
(seq-some #'1+ [2 4 6])                  ; 3
```

♥ `(seq-find predicate sequence &optional default)`
This function returns the first element in `sequence` for which `predicate` returns non-nil. If no element matches `predicate`, the function returns `default`.

```
(seq-find #'numberp ["abc" 1 nil])       ; 1
```

♥ `(seq-every-p predicate sequence)`
This function returns non-nil if applying `predicate` to every element of `sequence` returns non-nil.

```
(seq-every-p #'numberp [2 4 6])          ; t
(seq-every-p #'numberp [2 4 "6"])        ; nil
```

♥ `(seq-empty-p sequence)`

This function returns non-nil if `sequence` is empty.

♥ `(seq-count predicate sequence)`

This function returns the number of elements in `sequence` for which `predicate` returns non-nil.

```
(seq-count (lambda (elt) (> elt 0)) [-1 2 0 3 -1]) ; 2
```

♥ `(seq-sort function sequence)`

This function returns a copy of `sequence` that is sorted according to `function`, a function of two arguments that returns non-nil if the first argument should sort before the second.

```
(seq-sort #'> '(1 2 3 4))                ; (4 3 2 1)
```

♥ `(seq-sort-by function predicate sequence)`

This function is similar to `seq-sort`, but the elements of `sequence` are transformed by applying `function` on them before being sorted. `function` is a function of one argument.

```
(seq-sort-by #'seq-length #'> ["a" "ab" "abc"])
;;  ["abc" "ab" "a"]
```

♥ `(seq-contains-p sequence elt &optional function)`

This function returns non-nil if at least one element in `sequence` is equal to `elt`. If the optional argument `function` is non-nil, it is a function of two arguments to use instead of the default `equal`.

```
(seq-contains-p '(symbol1 symbol2) 'symbol1) ; t
(seq-contains-p '(symbol1 symbol2) 'symbol3) ; nil
```

♥ `(seq-set-equal-p sequence1 sequence2 &optional function)`

This function checks whether `sequence1` and `sequence2` contain the same elements, regardless of the order. If the optional argument `function` is non-nil, it is a function of two arguments to use instead of the default `equal`.

```
(seq-set-equal-p '(a b c) '(c b a))
;;  t
(seq-set-equal-p '(a b c) '(c b))
;;  nil
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a"))
;;  t
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a") #'eq)
;;  nil
```

♥ `(seq-position sequence elt &optional function)`

This function returns the index of the first element in `sequence` that is equal to `elt`. If the optional argument `function` is non-nil, it is a function of two arguments to use instead of the default `equal`.

```
(seq-position '(a b c) 'b)               ; 1
(seq-position '(a b c) 'd)               ; nil
```

♥ `(seq-uniq sequence &optional function)`

This function returns a list of the elements of `sequence` with duplicates removed. If the optional argument `function` is non-nil, it is a function of two arguments to use instead of the default `equal`.

```
(seq-uniq '(1 2 2 1 3))                 ; (1 2 3)
(seq-uniq '(1 2 2.0 1.0) #'=)           ; (1 2)
(seq-uniq '(1 2 2.0 1.0))               ; (1 2 2.0 1.0)
```

♥ `(seq-subseq sequence start &optional end)`

This function returns a subset of `sequence` from `start` to `end`, both integers (end defaults to the last element). If `start` or `end` is negative, it counts from the end of `sequence`.

```
(seq-subseq '(1 2 3 4 5) 1)
;;  (2 3 4 5)
(seq-subseq '[1 2 3 4 5] 1 3)
;;  [2 3]
(seq-subseq '[1 2 3 4 5] -3 -1)
;;  [3 4]
```

♥ `(seq-concatenate type &rest sequences)`

This function returns a sequence of type `type` made of the concatenation of `sequences`. `type` may be: vector, list or string.

```
(seq-concatenate 'list '(1 2) '(3 4) [5 6])
;;  (1 2 3 4 5 6)
(seq-concatenate 'string "Hello " "world")
;;  "Hello world"
```

♥ `(seq-mapcat function sequence &optional type)`

This function returns the result of applying `seq-concatenate` to the result of applying `function` to each element of `sequence`. The result is a sequence of type `type`, or a list if `type` is nil.

```
(seq-mapcat #'seq-reverse '((3 2 1) (6 5 4)))
;;  (1 2 3 4 5 6)
```

♥ `(seq-partition sequence n)`

This function returns a list of the elements of `sequence` grouped into sub-sequences of length `n`. The last sequence may contain less elements than `n`. `n` must be an integer. If `n` is a negative integer or 0, the return value is nil.

```
(seq-partition '(0 1 2 3 4 5 6 7) 3)
;;  ((0 1 2) (3 4 5) (6 7))
```

♥ `(seq-union sequence1 sequence2 &optional function)`

This function returns a list of the elements that appear either in `sequence1` or `sequence2`. The elements of the returned list are all unique, in the sense that no two elements there will compare equal. If the optional argument `function` is non-nil, it should be a function of two arguments to use to compare elements, instead of the default `equal`.

```
(seq-union [1 2 3] [3 5])
;;  (1 2 3 5)
```

♥ `(seq-intersection sequence1 sequence2 &optional function)`
This function returns a list of the elements that appear both in `sequence1` and `sequence2`. If the optional argument `function` is non-nil, it is a function of two arguments to use to compare elements instead of the default `equal`.

```
(seq-intersection [2 3 4 5] [1 3 5 6 7])
;; (3 5)
```

♥ `(seq-difference sequence1 sequence2 &optional function)`
This function returns a list of the elements that appear in `sequence1` but not in `sequence2`. If the optional argument `function` is non-nil, it is a function of two arguments to use to compare elements instead of the default `equal`.

```
(seq-difference '(2 3 4 5) [1 3 5 6 7])
;; (2 4)
```

♥ `(seq-group-by function sequence)`
This function separates the elements of `sequence` into an alist whose keys are the result of applying `function` to each element of `sequence`. Keys are compared using `equal`.

```
(seq-group-by #'integerp '(1 2.1 3 2 3.2))
;; ((t 1 3 2) (nil 2.1 3.2))
(seq-group-by #'car '((a 1) (b 2) (a 3) (c 4)))
;; ((b (b 2)) (a (a 1) (a 3)) (c (c 4)))
```

♥ `(seq-into sequence type)`
This function converts the sequence `sequence` into a sequence of type `type`. `type` can be one of the following symbols: vector, string or list.

```
(seq-into [1 2 3] 'list)
;; (1 2 3)
(seq-into nil 'vector)
;; []
(seq-into "hello" 'vector)
;; [104 101 108 108 111]
```

♥ `(seq-min sequence)`
This function returns the smallest element of `sequence`. The elements of `sequence` must be numbers or markers.

♥ `(seq-max sequence)`

♥ `(seq-doseq (var sequence)body...)`
This macro is like `dolist`, except that `sequence` can be a list, vector or string. This is primarily useful for side-effects.

♥ `(seq-let var-sequence val-sequence body...)`
This macro binds the variables defined in `var-sequence` to the values that are the corresponding elements of `val-sequence`. This is known as **destructuring binding**. The elements of `var-sequence` can themselves include sequences, allowing for nested destructuring.
The `var-sequence` sequence can also include the `&rest` marker followed by a variable name to be bound to the rest of `val-sequence`.

```
(seq-let [first second] [1 2 3 4]
  (list first second))
;;  (1 2)
(seq-let (_ a _ b) '(1 2 3 4)
  (list a b))
;;  (2 4)
(seq-let [a [b [c]]] [1 [2 [3]]]
  (list a b c))
;;  (1 2 3)
(seq-let [a b &rest others] [1 2 3 4]
  others)
;;  [3 4]
```

- `(seq-setq var-sequence val-sequence)`
  This macro works similarly to `seq-let`, except that values are assigned to variables as if by `setq` instead of as in a `let` binding.

```
(let ((a nil)
      (b nil))
  (seq-setq (_ a _ b) '(1 2 3 4))
  (list a b))
;;  (2 4)
```

- `(seq-random-elt sequence)`
  This function returns an element of `sequence` taken at random.

```
(seq-random-elt [1 2 3 4])                ; 3
(seq-random-elt [1 2 3 4])                ; 3
(seq-random-elt [1 2 3 4])                ; 3
(seq-random-elt [1 2 3 4])                ; 1
(seq-random-elt [1 2 3 4])                ; 2
```

6.2 Arrays

Emacs defines four types of array, all one-dimensional:

- strings
- vectors
- char-tables
- bool-vectors

Vectors and char-tables can hold elements of any type, but strings can only hold characters, and bool-vectors can only hold t and nil.

All four kinds of array share these characteristics:

- All have zero-origin indexing.
- The length of the array is fixed once you create it; you cannot change the length of an existing array.

- For purposes of evaluation, the array is a constant—i.e., it evaluates to itself.
- The elements of an array may be referenced or changed with the functions `aref` and `aset`, respectively

When you create an array, other than a char-table, you must specify its length. You cannot specify the length of a char-table, because that is determined by the range of character codes.

Here's the functions that accept all types of arrays.

- `(arrayp object)`
- `(aref arr index)`
  This function returns the `index`th element of the array or record `arr`.
- `(aset array index object)`
  This function sets the `index`th element of `array` to be object. It returns `object`.
- `(fillarray array object)`
  This function fills the array `array` with `object`, so that each element of `array` is `object`. It returns `array`.

```
(setq a (copy-sequence [a b c d e f g])) ; need to be mutable
;; [a b c d e f g]
(fillarray a 0)
;; [0 0 0 0 0 0 0]
a
;; [0 0 0 0 0 0 0]
```

6.3 Vectors

A vector is a general-purpose array whose elements can be any Lisp objects. Vectors are printed with square brackets surrounding the elements. You can write vectors in the same way in Lisp input.

Here are some functions that relate to vectors:

- `(vectorp object)`
- `(vector &rest objects)`
  This function creates and returns a vector whose elements are the arguments, `objects`.

```
(vector 'foo 23 [bar baz] "rats")
;; [foo 23 [bar baz] "rats"]
(vector)
;; []
```

- `(make-vector length object)`
  This function returns a new vector consisting of `length` elements, each initialized to `object`.

```
(setq sleepy (make-vector 9 'Z))
;; [Z Z Z Z Z Z Z Z Z]
```

- `(vconcat &rest sequences)`
  This function returns a new vector containing all the elements of `sequences`.

```
(setq a (vconcat '(A B C) '(D E F)))
;; [A B C D E F]
(eq a (vconcat a))
;; nil
(vconcat [A B C] "aa" '(foo (6 7)))
;; [A B C 97 97 foo (6 7)]
```

6.4 Char-Tables

A char-table is much like a vector, except that it is indexed by character codes. Any valid character code, without modifiers, can be used as an index in a char-table. In addition, a char-table can have extra slots to hold additional data not associated with particular character codes.

Each char-table has a **subtype** (a symbol) which serves two purposes:

- The subtype provides an easy way to tell what the char-table is for. For instance, display tables are char-tables with `display-table` as the subtype, and syntax tables are char-tables with `syntax-table` as the subtype.
- The subtype controls the number of extra slots in the char-table. This number is specified by the subtype's char-table-extra-slots symbol property, whose value should be integer between 0 and 10. If the subtype has no such symbol property, the char-table has no extra slots.

A char-table can have a parent, which is another char-table. If it does, then whenever the char-table specifies nil for a particular character c, it inherits the value specified in the parent.

A char-table can also have a default value. If so, then (aref char-table c) returns the default value whenever the char-table does not specify any other non-nil value.

- `(make-char-table subtype &optional init)`
  Return a newly-created char-table, with subtype `subtype` (a symbol). Each element is initialized to `init`, which defaults to nil. You cannot alter the subtype of a char-table after the char-table is created.
  There is no argument to specify the length of the char-table, because all char-tables have room for any valid character code as an index.
- `(char-table-p object)`
- `(char-table-subtype char-table)`
  This function returns the subtype symbol of `char-table`.
- `(char-table-parent char-table)`
  This function returns the parent of `char-table`.
- `(set-char-table-parent char-table new-parent)`
  This function sets the parent of `char-table` to `new-parent`.
- `(char-table-extra-slot char-table n)`
  This function returns the contents of extra slot `n` (zero based) of `char-table`.
- `(set-char-table-extra-slot char-table n value)`
  This function stores `value` in extra slot `n` (zero based) of `char-table`.

- `(char-table-range char-table range)`
  This returns the value specified in `char-table` for a range of characters `range`. Here are the possibilities for range:
  - nil: refers to the default value.
  - char: refers to the element for character char.
  - `(from . to)`: A cons cell refers to all the characters in the inclusive range `[from to]`.

- `(set-char-table-range char-table range value)`
  This function sets the value in `char-table` for a range of characters `range`. Here are the possibilities for `range`:
  - nil: refers to the default value.
  - t: refers to the whole range of character codes.
  - char: refers to the element for character char.
  - `(from . to)`: A cons cell refers to all the characters in the inclusive range `[from to]`.

- `(map-char-table function char-table)`
  This function calls `function` for each element of `char-table` that has a non-nil value. The call to `function` is with two arguments, a key and a value. The key is either a valid character or a cons cell `(from . to)`. The value is what `(char-table-range char-table key)` returns.
  The return value is always nil; to make calls to `map-char-table` useful, `function` should have side effects.

6.5 Bool-vectors

A bool-vector is much like a vector, except that it stores only the values t and nil. If you try to store any non-nil value into an element of the bool-vector, the effect is to store t there.

- `(make-bool-vector length initial)`
  Return a new bool-vector of `length` elements, each one initialized to `initial`.

- `(bool-vector &rest objects)`
  This function creates and returns a bool-vector whose elements are the arguments, `objects`.

- `(bool-vector-p object)`

- `bool-vector-exclusive-or a b &optional c`
  Return **bitwise exclusive or** of bool vectors `a` and `b`. If optional argument `c` is given, the result of this operation is stored into `c`. All arguments should be bool vectors of the same length.

- `(bool-vector-union a b &optional c)`
  Return **bitwise or** of bool vectors `a` and `b`.

- `(bool-vector-intersection a b &optional c)`
  Return **bitwise and** of bool vectors `a` and `b`.

- `(bool-vector-set-difference a b &optional c)`
  Return **set difference** of bool vectors `a` and `b`.

- `(bool-vector-not a &optional b)`
  Return **set complement** of bool vector `a`. If optional argument `b` is given, the result of this operation is

stored into b.

♥ `(bool-vector-subsetp a b)`
Return t if every t value in a is also t in b, nil otherwise.

♥ `(bool-vector-count-consecutive a b i)`
Return the number of consecutive elements in a equal b starting at i. a is a bool vector, b is t or nil, and i is an index into a.

♥ `(bool-vector-count-population a)`
Return the number of elements that are t in bool vector a.
The printed form represents up to 8 boolean values as a single character:

```
(bool-vector t nil t nil)                ; #&4"^E"
(bool-vector)                            ; #&0""

;; You can use vconcat to print a bool-vector like other vectors:
(vconcat (bool-vector nil t nil t))
;; [nil t nil t]
```

Chapter 7

Records

The purpose of records is to allow programmers to create objects with new types that are not built into Emacs. They are used as the underlying representation of `cl-defstruct` and `defclass` instances.

Internally, a record object is much like a vector; its slots can be accessed using `aref` and it can be copied using `copy-sequence`. However, the first slot is used to hold its type as returned by `type-of`.

The type slot should be a symbol or a type descriptor. If it's a type descriptor, the symbol naming its type will be returned. Any other kind of object is returned as-is.

The printed representation of records is `#s` followed by a list specifying the contents. The first list element must be the record type. The following elements are the record slots.

A record is considered a constant for evaluation: the result of evaluating it is the same record. This does not evaluate or even examine the slots.

♥ `(recordp object)`

♥ `(record type &rest objects)`

This function creates and returns a record whose type is `type` and remaining slots are the rest of the arguments, `objects`.

```
(record 'foo 23 [bar baz] "rats")
;; #s(foo 23 [bar baz] "rats")
```

♥ `(make-record type length object)`

This function returns a new record with type `type` and `length` more slots, each initialized to `object`.

```
(setq sleepy (make-record 'foo 9 'Z))
;; #s(foo Z Z Z Z Z Z Z Z Z)
```

Bibliography

[1] Aston Zhang et al. *Dive into Deep Learning*. 2021.

[2] Debra Cameron et al. *Learning GNU Emacs*. O'Reilly Media, Inc., 2004.

[3] Ian Goodfellow et al. *Deep Learning*. 2016.

[4] Yann LeCun et al. Gradientbased learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[5] D.E. Knuth. *The TEXbook*. Addison Wesley, 1986.

[6] D.E. Knuth. Typesetting concrete mathematics. *TUGboat*, 10(1):31–36, April 1989.

[7] Adrian Rosebrock. *Practical Python and OpenCV*. PyImageSearch, 2016.

[8] Adrian Rosebrock. *Deep Learning for Computer Vision with Python*. PyImageSearch, 2017.