0.1 New Features in the Lisp Language

- The new function `delete` is a traditional Lisp function. It takes two arguments, *elt* and *list*, and deletes from *list* any elements that are equal to *elt*. It uses the function `equal` to compare elements with *elt*.
- The new function `member` is a traditional Lisp function. It takes two arguments, *elt* and *list*, and finds the first element of *list* that is equal to *elt*. It uses the function `equal` to compare each list element with *elt*.

  The value is a sublist of *list*, whose first element is the one that was found. If no matching element is found, the value is `nil`.
- The new function `indirect-function` finds the effective function definition of an object called as a function. If the object is a symbol, `indirect-function` looks in the function definition of the symbol. It keeps doing this until it finds something that is not a symbol.
- There are new escape sequences for use in character and string constants. The escape sequence '`\a`' is equivalent to '`\C-g`', the ASCII BEL character (code 7). The escape sequence '`\x`' followed by a hexidecimal number represents the character whose ASCII code is that number. There is no limit on the number of digits in the hexidecimal value.
- The function `read` when reading from a buffer now does not skip a terminator character that terminates a symbol. It leaves that character to be read (or just skipped, if it is whitespace) next time.
- When you use a function *function* as the input stream for `read`, it is usually called with no arguments, and should return the next character. In Emacs 19, sometimes *function* is called with one argument (always a character). When that happens, *function* should save the argument and arrange to return it when called next time.
- `random` with integer argument n returns a random number between 0 and $n-1$.
- The functions `documentation` and `documentation-property` now take an additional optional argument which, if non-`nil`, says to refrain from calling `substitute-command-keys`. This way, you get the exact text of the documentation string as written, without the usual substitutions. Make sure to call `substitute-command-keys` yourself if you decide to display the string.
- The new function `invocation-name` returns as a string the program name that was used to run Emacs, with any directory names discarded.
- The new function `map-y-or-n-p` makes it convenient to ask a series of similar questions. The arguments are *prompter*, *actor*, *list*, and optional *help*.

  The value of *list* is a list of objects, or a function of no arguments to return either the next object or `nil` meaning there are no more.

  The argument *prompter* specifies how to ask each question. If *prompter* is a string, the question text is computed like this:

  ```
  (format prompter object)
  ```

  where *object* is the next object to ask about.

  If not a string, *prompter* should be a function of one argument (the next object to ask about) and should return the question text.

  The argument *actor* should be a function of one argument, which is called with each object that the user says yes for. Its argument is always one object from *list*.

If *help* is given, it is a list `(object objects action)`, where *object* is a string containing a singular noun that describes the objects conceptually being acted on; *objects* is the corresponding plural noun and *action* is a transitive verb describing *actor*. The default is `("object" "objects" "act on")`.

Each time a question is asked, the user may enter `y`, `Y`, or `SPC` to act on that object; `n`, `N`, or `DEL` to skip that object; `!` to act on all following objects; `ESC` or `q` to exit (skip all following objects); `.` (period) to act on the current object and then exit; or `C-h` to get help.

`map-y-or-n-p` returns the number of objects acted on.

- You can now "set" environment variables with the `setenv` command. This works by setting the variable `process-environment`, which `getenv` now examines in preference to the environment Emacs received from its parent.

0.2 New Features for Loading Libraries

You can now arrange to run a hook if a particular Lisp library is loaded.

The variable `after-load-alist` is an alist of expressions to be evalled when particular files are loaded. Each element looks like `(filename forms...)`.

When `load` is run and the file name argument equals *filename*, the *forms* in the corresponding element are executed at the end of loading. *filename* must match exactly! Normally *filename* is the name of a library, with no directory specified, since that is how `load` is normally called.

An error in *forms* does not undo the load, but does prevent execution of the rest of the *forms*.

The function `eval-after-load` provides a convenient way to add entries to the alist. Call it with two arguments, *file* and a form to execute.

The function `autoload` now supports autoloading a keymap. Use `keymap` as the fourth argument if the autoloaded function will become a keymap when loaded.

There is a new feature for specifying which functions in a library should be autoloaded by writing special "magic" comments in that library itself.

Write '`;;;###autoload`' on a line by itself before the real definition of the function, in its autoloadable source file; then the command `M-x update-file-autoloads` automatically puts the `autoload` call into `loaddefs.el`.

You can also put other kinds of forms into `loaddefs.el`, by writing '`;;;###autoload`' followed on the same line by the form. `M-x update-file-autoloads` copies the form from that line.

0.3 Compilation Features

- Inline functions.

  You can define an *inline function* with `defsubst`. Use `defsubst` just like `defun`, and it defines a function which you can call in all the usual ways. Whenever the function thus defined is used in compiled code, the compiler will open code it.

  You can get somewhat the same effects with a macro, but a macro has the limitation that you can use it only explicitly; a macro cannot be called with `apply`, `mapcar` and so

on. Also, it takes some work to convert an ordinary function into a macro. To convert it into an inline function, simply replace `defun` with `defsubst`.

Making a function inline makes explicit calls run faster. But it also has disadvantages. For one thing, it reduces flexibility; if you change the definition of the function, calls already inlined still use the old definition until you recompile them.

Another disadvantage is that making a large function inline can increase the size of compiled code both in files and in memory. Since the advantages of inline functions are greatest for small functions, you generally should not make large functions inline.

Inline functions can be used and open coded later on in the same file, following the definition, just like macros.

- The command `byte-compile-file` now offers to save any buffer visiting the file you are compiling.
- The new command `compile-defun` reads, compiles and executes the defun containing point. If you use this on a defun that is actually a function definition, the effect is to install a compiled version of that function.
- Whenever you load a Lisp file or library, you now receive a warning if the directory contains both a '`.el`' file and a '`.elc`' file, and the '`.el`' file is newer. This typically indicates that someone has updated the Lisp code but forgotten to recompile it, so the changes do not take effect. The warning is a reminder to recompile.
- The special form `eval-when-compile` marks the forms it contains to be evaluated at compile time *only*. At top-level, this is analogous to the Common Lisp idiom `(eval-when (compile) ...)`. Elsewhere, it is similar to the Common Lisp '`#.`' reader macro (but not when interpreting).

  If you're thinking of using this feature, we recommend you consider whether `provide` and `require` might do the job as well.
- The special form `eval-and-compile` is similar to `eval-when-compile`, but the whole form is evaluated both at compile time and at run time.

  If you're thinking of using this feature, we recommend you consider whether `provide` and `require` might do the job as well.
- Emacs Lisp has a new data type for byte-code functions. This makes them faster to call, and also saves space. Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears as a function to be called.

  The printed representation for a byte-code function object is like that for a vector, except that it starts with '`#`' before the opening '`[`'. A byte-code function object must have at least four elements; there is no maximum number, but only the first six elements are actually used. They are:

  arglist The list of argument symbols.

  byte-code The string containing the byte-code instructions.

  constants The vector of constants referenced by the byte code.

  stacksize The maximum stack size this function needs.

  docstring The documentation string (if any); otherwise, `nil`.

interactive
: The interactive spec (if any). This can be a string or a Lisp expression. It is `nil` for a function that isn't interactive.

The predicate `byte-code-function-p` tests whether a given object is a byte-code function.

You can create a byte-code function object in a Lisp program with the function `make-byte-code`. Its arguments are the elements to put in the byte-code function object.

You should not try to come up with the elements for a byte-code function yourself, because if they are inconsistent, Emacs may crash when you call the function. Always leave it to the byte compiler to create these objects; it, we hope, always makes the elements consistent.

0.4 Floating Point Numbers

You can now use floating point numbers in Emacs, if you define the macro `LISP_FLOAT_TYPE` when you compile Emacs.

The printed representation for floating point numbers requires either a decimal point surrounded by digits, or an exponent, or both. For example, '`1500.0`', '`15e2`', '`15.0e2`' and '`1.5e3`' are four ways of writing a floating point number whose value is 1500.

The existing predicate `numberp` now returns `t` if the argument is any kind of number—either integer or floating. The new predicates `integerp` and `floatp` check for specific types of numbers.

You can do arithmetic on floating point numbers with the ordinary arithmetic functions, `+`, `-`, `*` and `/`. If you call one of these functions with both integers and floating point numbers among the arguments, the arithmetic is done in floating point. The same applies to the numeric comparison functions such as `=` and `<`. The remainder function `%` does not accept floating point arguments, and neither do the bitwise boolean operations such as `logand` or the shift functions such as `ash`.

There is a new arithmetic function, `abs`, which returns the absolute value of its argument. It handles both integers and floating point numbers.

To convert an integer to floating point, use the function `float`. There are four functions to convert floating point numbers to integers; they differ in how they round. `truncate` rounds toward 0, `floor` rounds down, `ceil` rounds up, and `round` produces the nearest integer.

You can use `logb` to extract the binary exponent of a floating point number. More precisely, it is the logarithm base 2, rounded down to an integer.

Emacs has several new mathematical functions that accept any kind of number as argument, but always return floating point numbers.

`cos`
`sin`
`tan`
: Trigonometric functions.

`acos`
`asin`
`atan`
: Inverse trigonometric functions.

`exp` The exponential function (power of e).

`log` Logarithm base e.

`log10` Logarithm base 10

`expt` Raise x to power y.

`sqrt` The square root function.

The new function `string-to-number` now parses a string containing either an integer or a floating point number, returning the number.

The `format` function now handles the specifications '`%e`', '`%f`' and '`%g`' for printing floating point numbers; likewise `message`.

The new variable `float-output-format` controls how Lisp prints floating point numbers. Its value should be `nil` or a string.

If it is a string, it should contain a '`%`'-spec like those accepted by `printf` in C, but with some restrictions. It must start with the two characters '`%.`'. After that comes an integer which is the precision specification, and then a letter which controls the format.

The letters allowed are '`e`', '`f`' and '`g`'. Use '`e`' for exponential notation ('*`dig.digitseexpt`*'). Use '`f`' for decimal point notation ('*`digits.digits`*'). Use '`g`' to choose the shorter of those two formats for the number at hand.

The precision in any of these cases is the number of digits following the decimal point. With '`e`', a precision of 0 means to omit the decimal point. 0 is not allowed with '`f`' or '`g`'.

A value of `nil` means to use the format '`%.20g`'.

No matter what the value of `float-output-format`, printing ensures that the result fits the syntax rules for a floating point number. If it doesn't fit (for example, if it looks like an integer), it is modified to fit. By contrast, the `format` function formats floating point numbers without requiring the output to fit the syntax rules for floating point number.

0.5 New Features for Printing And Formatting Output

- The `format` function has a new feature: '`%S`'. This print spec prints any kind of Lisp object, even a string, using its Lisp printed representation.

  By contrast, '`%s`' prints everything without quotation.
- `prin1-to-string` now takes an optional second argument which says not to print the Lisp quotation characters. (In other words, to use `princ` instead of `prin1`.)
- The new variable `print-level` specifies the maximum depth of list nesting to print before cutting off all deeper structure. A value of `nil` means no limit.

0.6 Changes in Basic Editing Functions

- There are two new primitives for putting text in the kill ring: `kill-new` and `kill-append`.

  The function `kill-new` adds a string to the front of the kill ring.

  Use `kill-append` to add a string to a previous kill. The second argument *before-p*, if non-`nil`, says to add the string at the beginning; otherwise, it goes at the end.

  Both of these functions apply `interprogram-cut-function` to the entire string of killed text that ends up at the beginning of the kill ring.

- The new function `current-kill` rotates the yanking pointer in the kill ring by *n* places, and returns the text at that place in the ring. If the optional second argument *do-not-move* is non-`nil`, it doesn't actually move the yanking point; it just returns the *n*th kill forward. If *n* is zero, indicating a request for the latest kill, `current-kill` calls `interprogram-paste-function` (documented below) before consulting the kill ring.

  All Emacs Lisp programs should either use `current-kill`, `kill-new`, and `kill-append` to manipulate the kill ring, or be sure to call `interprogram-paste-function` and `interprogram-cut-function` as appropriate.
- The variables `interprogram-paste-function` and `interprogram-cut-function` exist so that you can provide functions to transfer killed text to and from other programs.
- The `kill-region` function can now be used in read-only buffers. It beeps, but adds the region to the kill ring without deleting it.
- The new function `compare-buffer-substrings` lets you compare two substrings of the same buffer or two different buffers. Its arguments look like this:

  `(compare-buffer-substrings` *`buf1 beg1 end1 buf2 beg2 end2`*`)`

  The first three arguments specify one substring, giving a buffer and two positions within the buffer. The last three arguments specify the other substring in the same way.

  The value is negative if the first substring is less, positive if the first is greater, and zero if they are equal. The absolute value of the result is one plus the index of the first different characters.
- Overwrite mode treats tab and newline characters specially. You can now turn off this special treatment by setting `overwrite-binary-mode` to `t`.
- Once the mark "exists" in a buffer, it normally never ceases to exist. However, in Transient Mark mode, it may become *inactive*. The variable `mark-active`, which is always local in all buffers, indicates whether the mark is active: non-`nil` means yes.

  When the mark is inactive, the function `mark` normally gets an error. However, `(mark t)` returns the position of the inactive mark.

  The function `push-mark` normally does not activate the mark. However, it accepts an optional third argument *activate* which, if non-`nil`, says to activate.

  A command can request deactivation of the mark upon return to the editor command loop by setting `deactivate-mark` to a non-`nil` value. Transient Mark mode works by causing the command loop to take note of `deactivate-mark` and actually deactivate the mark.

  Transient Mark mode enables highlighting of the region when the mark is active. This is currently implemented only under the X Window System. A few other commands vary their behavior slightly in this case, by testing `transient-mark-mode`. More specifically, they avoid special display actions such as moving the cursor temporarily, which are not needed when the region is shown by highlighting.

  The variables `activate-mark-hook` and `deactivate-mark-hook` are normal hooks run, respectively, when the mark becomes active and when it becomes inactive. The hook `activate-mark-hook` is also run at the end of a command if the mark is active and the region may have changed.
- The function `move-to-column` now accepts a second optional argument *force*, in addition to *column*; if the requested column *column* is in the middle of a tab character and

force is non-`nil`, `move-to-column` replaces the tab with the appropriate sequence of spaces so that it can place point exactly at *column*.

- The search functions when successful now return the value of point rather than just `t`. This affects the functions `search-forward`, `search-backward`, `word-search-forward`, `word-search-backward`, `re-search-forward`, and `re-search-backward`.
- When you do regular expression searching or matching, there is no longer a limit to how many '`\(...\)`' pairs you can get information about with `match-beginning` and `match-end`. Also, these parenthetical groupings may now be nested to any degree.
- In a regular expression, when you use an asterisk after a parenthetical grouping, and then ask about what range was matched by the grouping, Emacs 19 reports just its last occurrence. Emacs 18 used to report the range of all the repetitions put together.

  For example,

  ```
  (progn
   (string-match "f\\(o\\)*" "foo")
   (list (match-beginning 1)
         (match-end 1)))
  ```

  returns `(2 3)` in Emacs 19, corresponding to just the last repetition of '`\(o\)`'. In Emacs 18, that expression returns `(1 3)`, encompassing both repetitions.

  If you want the Emacs 18 behavior, use a grouping *containing* the asterisk: `"f\\(o*\\)"`.
- The new special form `save-match-data` preserves the regular expression match status. Usage: `(save-match-data` *`body...`*`)`.
- The function `translate-region` applies a translation table to the characters in a part of the buffer. Invoke it as `(translate-region` *`start end table`*`)`; *start* and *end* bound the region to translate.

  The translation table *table* is a string; `(aref` *`table ochar`*`)` gives the translated character corresponding to *ochar*. If the length of *table* is less than 256, any characters with codes larger than the length of *table* are not altered by the translation.

  `translate-region` returns the number of characters which were actually changed by the translation. This does not count characters which were mapped into themselves in the translation table.
- There are two new hook variables that let you notice all changes in all buffers (or in a particular buffer, if you make them buffer-local): `before-change-function` and `after-change-function`.

  If `before-change-function` is non-`nil`, then it is called before any buffer modification. Its arguments are the beginning and end of the region that is going to change, represented as integers. The buffer that's about to change is always the current buffer.

  If `after-change-function` is non-`nil`, then it is called after any buffer modification. It takes three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. (To get the current length, subtract the region beginning from the region end.) All three arguments are integers. The buffer that has just changed is always the current buffer.

  Both of these variables are temporarily bound to `nil` during the time that either of these hooks is running. This means that if one of these functions changes the buffer,

that change won't run these functions. If you do want hooks to be run recursively, write your hook functions to bind these variables back to their usual values.

- The hook `first-change-hook` is run using `run-hooks` whenever a buffer is changed that was previously in the unmodified state.
- The second argument to `insert-abbrev-table-description` is now optional.

0.7 Text Properties

Each character in a buffer or a string can have a *text property list*, much like the property list of a symbol. The properties belong to a particular character at a particular place, such as, the letter '`T`' at the beginning of this sentence. Each property has a name, which is usually a symbol, and an associated value, which can be any Lisp object—just as for properties of symbols.

You can use the property `face` to control the font and color of text. Several other property names have special meanings. You can create properties of any name and examine them later for your own purposes.

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as `substring`, `insert`, and `buffer-substring`.

Since text properties are considered part of the buffer contents, changing properties in a buffer "modifies" the buffer, and you can also undo such changes.

Strings with text properties have a special printed representation which describes all the properties. This representation is also the read syntax for such a string. It looks like this:

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

where *property-data* is zero or more elements in groups of three as follows:

```
beg end plist
```

The elements *beg* and *end* are integers, and together specify a portion of the string; *plist* is the property list for that portion.

0.7.1 Examining Text Properties

The simplest way to examine text properties is to ask for the value of a particular property of a particular character. For that, use `get-text-property`. Use `text-properties-at` to get the entire property list of a character.

`(get-text-property` *pos prop object*`)` returns the *prop* property of the character after *pos* in *object* (a buffer or string). The argument *object* is optional and defaults to the current buffer.

`(text-properties-at` *pos object*`)` returns the entire property list of the character after *pos* in the string or buffer *object* (which defaults to the current buffer).

0.7.2 Changing Text Properties

There are four primitives for changing properties of a specified range of text:

`add-text-properties`
: This function puts on specified properties, leaving other existing properties unaltered.

`put-text-property`
: This function puts on a single specified property, leaving others unaltered.

`remove-text-properties`
: This function removes specified properties, leaving other properties unaltered.

`set-text-properties`
: This function replaces the entire property list, leaving no vestige of the properties that that text used to have.

All these functions take four arguments: *start*, *end*, *props*, and *object*. The last argument is optional and defaults to the current buffer. The argument *props* has the form of a property list.

0.7.3 Property Search Functions

In typical use of text properties, most of the time several or many consecutive characters have the same value for a property. Rather than writing your programs to examine characters one by one, it is much faster to process chunks of text that have the same property value.

The functions `next-property-change` and `previous-property-change` scan forward or backward from position *pos* in *object*, looking for a change in any property between two characters scanned. They returns the position between those two characters, or `nil` if no change is found.

The functions `next-single-property-change` and `previous-single-property-change` are similar except that you specify a particular property and they look for changes in the value of that property only. The property is the second argument, and *object* is third.

0.7.4 Special Properties

If a character has a `category` property, we call it the *category* of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

You can use the property `face` to control the font and color of text.

You can specify a different keymap for a portion of the text by means of a `local-map` property. The property's value, for the character after point, replaces the buffer's local map.

If a character has the property `read-only`, then modifying that character is not allowed. Any command that would do so gets an error.

If a character has the property `modification-hooks`, then its value should be a list of functions; modifying that character calls all of those functions. Each function receives two arguments: the beginning and end of the part of the buffer being modified. Note that if a particular modification hook function appears on several characters being modified by a single primitive, you can't predict how many times the function will be called.

Insertion of text does not, strictly speaking, change any existing character, so there is a special rule for insertion. It compares the `read-only` properties of the two surrounding characters; if they are `eq`, then the insertion is not allowed. Assuming insertion is allowed, it then gets the `modification-hooks` properties of those characters and calls all the functions in each of them. (If a function appears on both characters, it may be called once or twice.)

The special properties `point-entered` and `point-left` record hook functions that report motion of point. Each time point moves, Emacs compares these two property values:

- the `point-left` property of the character after the old location, and
- the `point-entered` property of the character after the new location.

If these two values differ, each of them is called (if not `nil`) with two arguments: the old value of point, and the new one.

The same comparison is made for the characters before the old and new locations. The result may be to execute two `point-left` functions (which may be the same function) and/or two `point-entered` functions (which may be the same function). The `point-left` functions are always called before the `point-entered` functions.

A primitive function may examine characters at various positions without moving point to those positions. Only an actual change in the value of point runs these hook functions.

0.8 New Features for Files

- The new function `file-accessible-directory-p` tells you whether you can open files in a particular directory. Specify as an argument either a directory name or a file name which names a directory file. The function returns `t` if you can open existing files in that directory.
- The new function `file-executable-p` returns `t` if its argument is the name of a file you have permission to execute.
- The function `file-truename` returns the "true name" of a specified file. This is the name that you get by following symbolic links until none remain. The argument must be an absolute file name.
- New functions `make-directory` and `delete-directory` create and delete directories. They both take one argument, which is the name of the directory as a file.
- The function `read-file-name` now takes an additional argument which specifies an initial file name. If you specify this argument, `read-file-name` inserts it along with the directory name. It puts the cursor between the directory and the initial file name.

  The user can then use the initial file name unchanged, modify it, or simply kill it with `C-k`.

  If the variable `insert-default-directory` is `nil`, then the default directory is not inserted, and the new argument is ignored.
- The function `file-relative-name` does the inverse of expansion—it tries to return a relative name which is equivalent to *filename* when interpreted relative to *directory*. (If such a relative name would be longer than the absolute name, it returns the absolute name instead.)
- The function `file-newest-backup` returns the name of the most recent backup file for *filename*, or `nil` that file has no backup files.
- The list returned by `file-attributes` now has 12 elements. The 12th element is the file system number of the file system that the file is in. This element together with the file's inode number, which is the 11th element, give enough information to distinguish any two files on the system—no two files can have the same values for both of these numbers.

- The new function `set-visited-file-modtime` updates the current buffer's recorded modification time from the visited file's time.

  This is useful if the buffer was not read from the file normally, or if the file itself has been changed for some known benign reason.

  If you give the function an argument, that argument specifies the new value for the recorded modification time. The argument should be a list of the form `(`*`high . low`*`)` or `(`*`high low`*`)` containing two integers, each of which holds 16 bits of the time. (This is the same format that `file-attributes` uses to return time values.)

  The new function `visited-file-modtime` returns the recorded last modification time, in that same format.

- The function `directory-files` now takes an optional fourth argument which, if non-`nil`, inhibits sorting the file names. Use this if you want the utmost possible speed and don't care what order the files are processed in.

  If the order of processing is at all visible to the user, then the user will probably be happier if you do sort the names.

- The variable `directory-abbrev-alist` contains an alist of abbreviations to use for file directories. Each element has the form `(`*`from . to`*`)`, and says to replace *from* with *to* when it appears in a directory name. This replacement is done when setting up the default directory of a newly visited file. The *from* string is actually a regular expression; it should always start with '`^`'.

  You can set this variable in `site-init.el` to describe the abbreviations appropriate for your site.

- The function `abbreviate-file-name` applies abbreviations from `directory-abbrev-alist` to its argument, and substitutes '`~`' for the user's home directory.

  Abbreviated directory names are useful for directories that are normally accessed through symbolic links. If you think of the link's name as "the name" of the directory, you can define it as an abbreviation for the directory's official name; then ordinarily Emacs will call that directory by the link name you normally use.

- `write-region` can write a given string instead of text from the buffer. Use the string as the first argument (in place of the starting character position).

  You can supply a second file name as the fifth argument (*visit*). Use this to write the data to one file (the first argument, *filename*) while nominally visiting a different file (the fifth argument, *visit*). The argument *visit* is used in the echo area message and also for file locking; *visit* is stored in `buffer-file-name`.

- The value of `write-file-hooks` does not change when you switch to a new major mode. The intention is that these hooks have to do with where the file came from, and not with what it contains.

- There is a new hook variable for saving files: `write-contents-hooks`. It works just like `write-file-hooks` except that switching to a new major mode clears it back to `nil`. Major modes should use this hook variable rather than `write-file-hooks`.

- The hook `after-save-hook` runs just after a buffer has been saved in its visited file.

- The new function `set-default-file-modes` sets the file protection for new files created with Emacs. The argument must be an integer. (It would be better to permit symbolic

arguments like the `chmod` program, but that would take more work than this function merits.)

Use the new function `default-file-modes` to read the current default file mode.

- Call the new function `unix-sync` to force all pending disk output to happen as soon as possible.

0.9 Making Certain File Names "Magic"

You can implement special handling for a class of file names. You must supply a regular expression to define the class of names (all those which match the regular expression), plus a handler that implements all the primitive Emacs file operations for file names that do match.

The value of `file-name-handler-alist` is a list of handlers, together with regular expressions that decide when to apply each handler. Each element has the form `(`*`regexp . handler`*`)`. If a file name matches *regexp*, then all work on that file is done by calling *handler*.

All the Emacs primitives for file access and file name transformation check the given file name against `file-name-handler-alist`, and call *handler* to do the work if appropriate. The first argument given to *handler* is the name of the primitive; the remaining arguments are the arguments that were passed to that primitive. (The first of these arguments is typically the file name itself.) For example, if you do this:

```
(file-exists-p filename)
```

and *filename* has handler *handler*, then *handler* is called like this:

```
(funcall handler 'file-exists-p filename)
```

Here are the primitives that you can handle in this way:

`add-name-to-file`, `copy-file`, `delete-directory`, `delete-file`, `directory-file-name`, `directory-files`, `dired-compress-file`, `dired-uncache`, `expand-file-name`, `file-accessible-directory-p`, `file-attributes`, `file-directory-p`, `file-executable-p`, `file-exists-p`, `file-local-copy`, `file-modes`, `file-name-all-completions`, `file-name-as-directory`, `file-name-completion`, `file-name-directory`, `file-name-nondirectory`, `file-name-sans-versions`, `file-newer-than-file-p`, `file-readable-p`, `file-symlink-p`, `file-writable-p`, `insert-directory`, `insert-file-contents`, `load`, `make-directory`, `make-symbolic-link`, `rename-file`, `set-file-modes`, `set-visited-file-modtime`, `unhandled-file-name-directory`, `verify-visited-file-modtime`, `write-region`.

The handler function must handle all of the above operations, and possibly others to be added in the future. Therefore, it should always reinvoke the ordinary Lisp primitive when it receives an operation it does not recognize. Here's one way to do this:

```
(defun my-file-handler (operation &rest args)
  ;; First check for the specific operations
  ;; that we have special handling for.
  (cond ((eq operation 'insert-file-contents) ...)
        ((eq operation 'write-region) ...)
        ...
        ;; Handle any operation we don't know about.
```

```
(t (let (file-name-handler-alist)
     (apply operation args)))))
```

The function `file-local-copy` copies file *filename* to the local site, if it isn't there already. If *filename* specifies a "magic" file name which programs outside Emacs cannot directly read or write, this copies the contents to an ordinary file and returns that file's name.

If *filename* is an ordinary file name, not magic, then this function does nothing and returns `nil`.

The function `unhandled-file-name-directory` is used to get a non-magic directory name from an arbitrary file name. It uses the directory part of the specified file name if that is not magic. Otherwise, it asks the file name's handler what to do.

0.10 Frames

Emacs now supports multiple X windows via a new data type known as a *frame*.

A frame is a rectangle on the screen that contains one or more Emacs windows. Subdividing a frame works just like subdividing the screen in earlier versions of Emacs.

There are two kinds of frames: terminal frames and X window frames. Emacs creates one terminal frame when it starts up with no X display; it uses Termcap or Terminfo to display using characters. There is no way to create another terminal frame after startup. If Emacs has an X display, it does not make a terminal frame, and there is none.

When you are using X windows, Emacs starts out with a single X window frame. You can create any number of X window frames using `make-frame`.

Use the predicate `framep` to determine whether a given Lisp object is a frame.

The function `redraw-frame` redisplays the entire contents of a given frame.

0.10.1 Creating and Deleting Frames

Use `make-frame` to create a new frame. This is the only primitive for creating frames. In principle it could work under any window system which Emacs understands; the only one we support is X.

`make-frame` takes just one argument, which is an alist specifying frame parameters. Any parameters not mentioned in the argument alist default based on the value of `default-frame-alist`; parameters not specified there default from the standard X defaults file and X resources.

When you invoke Emacs, if you specify arguments for window appearance and so forth, these go into `default-frame-alist` and that is how they have their effect.

You can specify the parameters for the initial startup X window frame by setting `initial-frame-alist` in your `.emacs` file. If these parameters specify a separate minibuffer-only frame, and you have not created one, Emacs creates one for you, using the parameter values specified in `minibuffer-frame-alist`.

You can specify the size and position of a frame using the frame parameters `left`, `top`, `height` and `width`. You must specify either both size parameters or neither. You must specify either both position parameters or neither. The geometry parameters that you don't specify are chosen by the window manager in its usual fashion.

The function `x-parse-geometry` converts a standard X-style geometry string to an alist which you can use as part of the argument to `make-frame`.

Use the function `delete-frame` to eliminate a frame. Frames are like buffers where deletion is concerned; a frame actually continues to exist as a Lisp object until it is deleted *and* there are no references to it, but once it is deleted, it has no further effect on the screen.

The function `frame-live-p` returns non-`nil` if the argument (a frame) has not been deleted.

0.10.2 Finding All Frames

The function `frame-list` returns a list of all the frames that have not been deleted. It is analogous to `buffer-list`. The list that you get is newly created, so modifying the list doesn't have any effect on the internals of Emacs. The function `visible-frame-list` returns the list of just the frames that are visible.

`next-frame` lets you cycle conveniently through all the frames from an arbitrary starting point. Its first argument is a frame. Its second argument *minibuf* says what to do about minibuffers:

`nil` Exclude minibuffer-only frames.

a window Consider only the frames using that particular window as their minibuffer.

anything else
Consider all frames.

0.10.3 Frames and Windows

All the non-minibuffer windows in a frame are arranged in a tree of subdivisions; the root of this tree is available via the function `frame-root-window`. Each window is part of one and only one frame; you can get the frame with `window-frame`.

At any time, exactly one window on any frame is *selected within the frame*. You can get the frame's current selected window with `frame-selected-window`. The significance of this designation is that selecting the frame selects for Emacs as a whole the window currently selected within that frame.

Conversely, selecting a window for Emacs with `select-window` also makes that window selected within its frame.

0.10.4 Frame Visibility

A frame may be *visible*, *invisible*, or *iconified*. If it is invisible, it doesn't show in the screen, not even as an icon. You can set the visibility status of a frame with `make-frame-visible`, `make-frame-invisible`, and `iconify-frame`. You can examine the visibility status with `frame-visible-p`—it returns `t` for a visible frame, `nil` for an invisible frame, and `icon` for an iconified frame.

0.10.5 Selected Frame

At any time, one frame in Emacs is the *selected frame*. The selected window always resides on the selected frame.

`selected-frame` [Function]
This function returns the selected frame.

The X server normally directs keyboard input to the X window that the mouse is in. Some window managers use mouse clicks or keyboard events to *shift the focus* to various X windows, overriding the normal behavior of the server.

Lisp programs can switch frames "temporarily" by calling the function `select-frame`. This does not override the window manager; rather, it escapes from the window manager's control until that control is somehow reasserted. The function takes one argument, a frame, and selects that frame. The selection lasts until the next time the user does something to select a different frame, or until the next time this function is called.

Emacs cooperates with the X server and the window managers by arranging to select frames according to what the server and window manager ask for. It does so by generating a special kind of input event, called a *focus* event. The command loop handles a focus event by calling `internal-select-frame`.

0.10.6 Frame Size and Position

The new functions `frame-height` and `frame-width` return the height and width of a specified frame (or of the selected frame), measured in characters.

The new functions `frame-pixel-height` and `frame-pixel-width` return the height and width of a specified frame (or of the selected frame), measured in pixels.

The new functions `frame-char-height` and `frame-char-width` return the height and width of a character in a specified frame (or in the selected frame), measured in pixels.

`set-frame-size` sets the size of a frame, measured in characters; its arguments are *frame*, *cols* and *rows*. To set the size with values measured in pixels, you can use `modify-frame-parameters`.

The function `set-frame-position` sets the position of the top left corner of a frame. Its arguments are *frame*, *left* and *top*.

0.10.7 Frame Parameters

A frame has many parameters that affect how it displays. Use the function `frame-parameters` to get an alist of all the parameters of a given frame. To alter parameters, use `modify-frame-parameters`, which takes two arguments: the frame to modify, and an alist of parameters to change and their new values. Each element of *alist* has the form `(`*`parm`*` . `*`value`*`)`, where *parm* is a symbol. Parameters that aren't meaningful are ignored. If you don't mention a parameter in *alist*, its value doesn't change.

Just what parameters a frame has depends on what display mechanism it uses. Here is a table of the parameters of an X window frame:

`name` The name of the frame.

`left` The screen position of the left edge.

`top` The screen position of the top edge.

`height` The height of the frame contents, in pixels.

`width` The width of the frame contents, in pixels.

`window-id`
The number of the X window for the frame.

`minibuffer`
: Whether this frame has its own minibuffer. `t` means yes, `none` means no, `only` means this frame is just a minibuffer, a minibuffer window (in some other frame) means the new frame uses that minibuffer.

`font`
: The name of the font for the text.

`foreground-color`
: The color to use for the inside of a character. Use strings to designate colors; the X server defines the meaningful color names.

`background-color`
: The color to use for the background of text.

`mouse-color`
: The color for the mouse cursor.

`cursor-color`
: The color for the cursor that shows point.

`border-color`
: The color for the border of the frame.

`cursor-type`
: The way to display the cursor. There are two legitimate values: `bar` and `box`. The value `bar` specifies a vertical bar between characters as the cursor. The value `box` specifies an ordinary black box overlaying the character after point; that is the default.

`icon-type`
: Non-`nil` for a bitmap icon, `nil` for a text icon.

`border-width`
: The width in pixels of the window border.

`internal-border-width`
: The distance in pixels between text and border.

`auto-raise`
: Non-`nil` means selecting the frame raises it.

`auto-lower`
: Non-`nil` means deselecting the frame lowers it.

`vertical-scroll-bars`
: Non-`nil` gives the frame a scroll bar for vertical scrolling.

0.10.8 Minibufferless Frames

Normally, each frame has its own minibuffer window at the bottom, which is used whenever that frame is selected. However, you can also create frames with no minibuffers. These frames must use the minibuffer window of some other frame.

The variable `default-minibuffer-frame` specifies where to find a minibuffer for frames created without minibuffers of their own. Its value should be a frame which does have a minibuffer.

You can also specify a minibuffer window explicitly when you create a frame; then `default-minibuffer-frame` is not used.

0.11 X Window System Features

- The new functions `mouse-position` and `set-mouse-position` give access to the current position of the mouse.

  `mouse-position` returns a description of the position of the mouse. The value looks like `(`*frame* `x . y)`, where *x* and *y* are measured in pixels relative to the top left corner of the inside of *frame*.

  `set-mouse-position` takes three arguments, *frame*, *x* and *y*, and warps the mouse cursor to that location on the screen.

- `track-mouse` is a new special form for tracking mouse motion. Use it in definitions of mouse clicks that want pay to attention to the motion of the mouse, not just where the buttons are pressed and released. Here is how to use it:

  ```
  (track-mouse body...)
  ```

  While *body* executes, mouse motion generates input events just as mouse clicks do. *body* can read them with `read-event` or `read-key-sequence`.

  `track-mouse` returns the value of the last form in *body*.

  The format of these events is described under "New Input Event Formats."

- `x-set-selection` sets a "selection" in the X server. It takes two arguments: a selection type *type*, and the value to assign to it, *data*. If *data* is `nil`, it means to clear out the selection. Otherwise, *data* may be a string, a symbol, an integer (or a cons of two integers or list of two integers), or a cons of two markers pointing to the same buffer. In the last case, the selection is considered to be the text between the markers. The data may also be a vector of valid non-vector selection values.

  Each possible *type* has its own selection value, which changes independently. The usual values of *type* are `PRIMARY` and `SECONDARY`; these are symbols with upper-case names, in accord with X protocol conventions. The default is `PRIMARY`.

  To get the value of the selection, call `x-get-selection`. This function accesses selections set up by Emacs and those set up by other X clients. It takes two optional arguments, *type* and *data-type*. The default for *type* is `PRIMARY`.

  The *data-type* argument specifies the form of data conversion to use; meaningful values include `TEXT`, `STRING`, `TARGETS`, `LENGTH`, `DELETE`, `FILE_NAME`, `CHARACTER_POSITION`, `LINE_NUMBER`, `COLUMN_NUMBER`, `OWNER_OS`, `HOST_NAME`, `USER`, `CLASS`, `NAME`, `ATOM`, and `INTEGER`. (These are symbols with upper-case names in accord with X Windows conventions.) The default for *data-type* is `STRING`.

- The X server has a set of numbered *cut buffers* which can store text or other data being moved between applications. Use `x-get-cut-buffer` to get the contents of a cut buffer; specify the cut buffer number as argument. Use `x-set-cut-buffer` with argument *string* to store a new string into the first cut buffer (moving the other values down through the series of cut buffers, kill-ring-style).

  Cut buffers are considered obsolete, but Emacs supports them for the sake of X clients that still use them.

- You can close the connection with the X server with the function `x-close-current-connection`. This takes no arguments.

  Then you can connect to a different X server with `x-open-connection`. The first argument, *display*, is the name of the display to connect to.

  The optional second argument *xrm-string* is a string of resource names and values, in the same format used in the `.Xresources` file. The values you specify override the resource values recorded in the X server itself. Here's an example of what this string might look like:

  ```
  "*BorderWidth: 3\n*InternalBorder: 2\n"
  ```

- A series of new functions give you information about the X server and the screen you are using.

  `x-display-screens`
  : The number of screens associated with the current display.

  `x-server-version`
  : The version numbers of the X server in use.

  `x-server-vendor`
  : The vendor supporting the X server in use.

  `x-display-pixel-height`
  : The height of this X screen in pixels.

  `x-display-mm-height`
  : The height of this X screen in millimeters.

  `x-display-pixel-width`
  : The width of this X screen in pixels.

  `x-display-mm-width`
  : The width of this X screen in millimeters.

  `x-display-backing-store`
  : The backing store capability of this screen. Values can be the symbols `always`, `when-mapped`, or `not-useful`.

  `x-display-save-under`
  : Non-`nil` if this X screen supports the SaveUnder feature.

  `x-display-planes`
  : The number of planes this display supports.

  `x-display-visual-class`
  : The visual class for this X screen. The value is one of the symbols `static-gray`, `gray-scale`, `static-color`, `pseudo-color`, `true-color`, and `direct-color`.

  `x-display-color-p`
  : `t` if the X screen in use is a color screen.

  `x-display-color-cells`
  : The number of color cells this X screen supports.

There is also a variable `x-no-window-manager`, whose value is `t` if no X window manager is in use.

- The function `x-synchronize` enables or disables an X Windows debugging mode: synchronous communication. It takes one argument, non-`nil` to enable the mode and `nil` to disable.

  In synchronous mode, Emacs waits for a response to each X protocol command before doing anything else. This means that errors are reported right away, and you can directly find the erroneous command. Synchronous mode is not the default because it is much slower.

- The function `x-get-resource` retrieves a resource value from the X Windows defaults database. Its three arguments are *attribute*, *name* and *class*. It searches using a key of the form '*`instance.attribute`*', with class '`Emacs`', where *instance* is the name under which Emacs was invoked.

  The optional arguments *component* and *subclass* add to the key and the class, respectively. You must specify both of them or neither. If you specify them, the key is '*`instance.component.attribute`*', and the class is '`Emacs.`*`subclass`*'.

- `x-display-color-p` returns `t` if you are using an X server with a color display, and `nil` otherwise.

  `x-color-defined-p` takes as argument a string describing a color; it returns `t` if the display supports that color. (If the color is `"black"` or `"white"` then even black-and-white displays support it.)

- `x-popup-menu` has been generalized. It now accepts a keymap as the *menu* argument. Then the menu items are the prompt strings of individual key bindings, and the item values are the keys which have those bindings.

  You can also supply a list of keymaps as the first argument; then each keymap makes one menu pane (but keymaps that don't provide any menu items don't appear in the menu at all).

  `x-popup-menu` also accepts a mouse button event as the *position* argument. Then it displays the menu at the location at which the event took place. This is convenient for mouse-invoked commands that pop up menus.

- You can use the function `x-rebind-key` to change the sequence of characters generated by the X server for one of the keyboard keys.

  The first two arguments, *keycode* and *shift-mask*, should be numbers representing the keyboard code and shift mask respectively. They specify what key to change.

  The third argument, *newstring*, is the new definition of the key. It is a sequence of characters that the key should produce as input.

  The shift mask value is a combination of bits according to this table:

8	Control
4	Meta
2	Shift
1	Shift Lock

If you specify `nil` for *shift-mask*, then the key specified by *keycode* is redefined for all possible shift combinations.

For the possible values of *keycode* and their meanings, see the file `/usr/lib/Xkeymap.txt`. Keep in mind that the codes in that file are in octal!

The related function `x-rebind-keys` redefines a single keyboard key, specifying the behavior for each of the 16 shift masks independently. The first argument is *keycode*, as in `x-rebind-key`. The second argument *strings* is a list of 16 elements, one for each possible shift mask value; each element says how to redefine the key *keycode* with the corresponding shift mask value. If an element is a string, it is the new definition. If an element is `nil`, the definition does not change for that shift mask.

- The function `x-parse-geometry` parses a string specifying window size and position in the usual X format. It returns an alist describing which parameters were specified, and the values that were given for them.

  The elements of the alist look like (*parameter* . *value*). The possible *parameter* values are `left`, `top`, `width`, and `height`.

0.12 New Window Features

- The new function `window-at` tells you which window contains a given horizontal and vertical position on a specified frame. Call it with three arguments, like this:

  ```
  (window-at x column frame)
  ```

  The function returns the window which contains that cursor position in the frame *frame*. If you omit *frame*, the selected frame is used.

- The function `coordinates-in-window-p` takes two arguments and checks whether a particular frame position falls within a particular window.

  ```
  (coordinates-in-window-p coordinates window)
  ```

  The argument *coordinates* is a cons cell of this form:

  ```
  (x . y)
  ```

  The two coordinates are measured in characters, and count from the top left corner of the screen or frame.

  The value of the function tells you what part of the window the position is in. The possible values are:

  (*relx* . *rely*)
  : The coordinates are inside *window*. The numbers *relx* and *rely* are equivalent window-relative coordinates, counting from 0 at the top left corner of the window.

  `mode-line`
  : The coordinates are in the mode line of *window*.

  `vertical-split`
  : The coordinates are in the vertical line between *window* and its neighbor to the right.

  `nil`
  : The coordinates are not in any sense within *window*.

You need not specify a frame when you call `coordinates-in-window-p`, because it assumes you mean the frame which window *window* is on.

- The function `minibuffer-window` now accepts a frame as argument and returns the minibuffer window used for that frame. If you don't specify a frame, the currently selected frame is used. The minibuffer window may be on the frame in question, but if that frame has no minibuffer of its own, it uses the minibuffer window of some other frame, and `minibuffer-window` returns that window.
- Use `window-live-p` to test whether a window is still alive (that is, not deleted).
- Use `window-minibuffer-p` to determine whether a given window is a minibuffer or not. It no longer works to do this by comparing the window with the result of `(minibuffer-window)`, because there can be more than one minibuffer window at a time (if you have multiple frames).
- If you set the variable `pop-up-frames` non-`nil`, then the functions to show something "in another window" actually create a new frame for the new window. Thus, you will tend to have a frame for each window, and you can easily have a frame for each buffer.

  The value of the variable `pop-up-frame-function` controls how new frames are made. The value should be a function which takes no arguments and returns a frame. The default value is a function which creates a frame using parameters from `pop-up-frame-alist`.
- `display-buffer` is the basic primitive for finding a way to show a buffer on the screen. You can customize its behavior by storing a function in the variable `display-buffer-function`. If this variable is non-`nil`, then `display-buffer` calls it to do the work. Your function should accept two arguments, as follows:

  buffer — The buffer to be displayed.

  flag — A flag which, if non-`nil`, means you should find another window to display *buffer* in, even if it is already visible in the selected window.

  The function you supply will be used by commands such as `switch-to-buffer-other-window` and `find-file-other-window` as well as for your own calls to `display-buffer`.
- `delete-window` now gives all of the deleted window's screen space to a single neighboring window. Likewise, `enlarge-window` takes space from only one neighboring window until that window disappears; only then does it take from another window.
- `next-window` and `previous-window` accept another argument, *all-frames*.

  These functions now take three optional arguments: *window*, *minibuf* and *all-frames*. *window* is the window to start from (`nil` means use the selected window). *minibuf* says whether to include the minibuffer in the windows to cycle through: `t` means yes, `nil` means yes if it is active, and anything else means no.

  Normally, these functions cycle through all the windows in the selected frame, plus the minibuffer used by the selected frame even if it lies in some other frame.

  If *all-frames* is `t`, then these functions cycle through all the windows in all the frames that currently exist. If *all-frames* is neither `t` nor `nil`, then they limit themselves strictly to the windows in the selected frame, excluding the minibuffer in use if it lies in some other frame.

- The functions `get-lru-window` and `get-largest-window` now take an optional argument *all-frames*. If it is non-`nil`, the functions consider all windows on all frames. Otherwise, they consider just the windows on the selected frame.

  Likewise, `get-buffer-window` takes an optional second argument *all-frames*.

- The variable `other-window-scroll-buffer` specifies which buffer `scroll-other-window` should scroll.

- You can now mark a window as "dedicated" to its buffer. Then Emacs will not try to use that window for any other buffer unless you explicitly request it.

  Use the new function `set-window-dedicated-p` to set the dedication flag of a window *window* to the value *flag*. If *flag* is `t`, this makes the window dedicated. If *flag* is `nil`, this makes the window non-dedicated.

  Use `window-dedicated-p` to examine the dedication flag of a specified window.

- The new function `walk-windows` cycles through all visible windows, calling `proc` once for each window with the window as its sole argument.

  The optional second argument *minibuf* says whether to include minibuffer windows. A value of `t` means count the minibuffer window even if not active. A value of `nil` means count it only if active. Any other value means not to count the minibuffer even if it is active.

  If the optional third argument *all-frames* is `t`, that means include all windows in all frames. If *all-frames* is `nil`, it means to cycle within the selected frame, but include the minibuffer window (if *minibuf* says so) that that frame uses, even if it is on another frame. If *all-frames* is neither `nil` nor `t`, `walk-windows` sticks strictly to the selected frame.

- The function `window-end` is a counterpart to `window-start`: it returns the buffer position of the end of the display in a given window (or the selected window).

- The function `window-configuration-p` returns non-`nil` when given an object that is a window configuration (such as is returned by `current-window-configuration`).

0.13 Display Features

- `baud-rate` is now a variable rather than a function. This is so you can set it to reflect the effective speed of your terminal, when the system doesn't accurately know the speed.

- You can now remove any echo area message and make the minibuffer visible. To do this, call `message` with `nil` as the only argument. This clears any existing message, and lets the current minibuffer contents show through. Previously, there was no reliable way to make sure that the minibuffer contents were visible.

- The variable `temp-buffer-show-hook` has been renamed `temp-buffer-show-function`, because its value is a single function (of one argument), not a normal hook.

- The new function `force-mode-line-update` causes redisplay of the current buffer's mode line.

0.14 Display Tables

You can use the *display table* feature to control how all 256 possible character codes display on the screen. This is useful for displaying European languages that have letters not in the ASCII character set.

The display table maps each character code into a sequence of *glyphs*, each glyph being an image that takes up one character position on the screen. You can also define how to display each glyph on your terminal, using the *glyph table*.

0.14.1 Display Tables Proper

Use `make-display-table` to create a display table. The table initially has `nil` in all elements.

A display table is actually an array of 261 elements. The first 256 elements of a display table control how to display each possible text character. The value should be `nil` or a vector (which is a sequence of glyphs; see below). `nil` as an element means to display that character following the usual display conventions.

The remaining five elements of a display table serve special purposes (`nil` means use the default stated below):

256 The glyph for the end of a truncated screen line (the default for this is '\').

257 The glyph for the end of a continued line (the default is '`$`').

258 The glyph for the indicating an octal character code (the default is '\').

259 The glyph for indicating a control characters (the default is '`^`').

260 The vector of glyphs for indicating the presence of invisible lines (the default is '`...`').

Each buffer typically has its own display table. The display table for the current buffer is stored in `buffer-display-table`. (This variable automatically becomes local if you set it.) If this variable is `nil`, the value of `standard-display-table` is used in that buffer.

Each window can have its own display table, which overrides the display table of the buffer it is showing.

If neither the selected window nor the current buffer has a display table, and if `standard-display-table` is `nil`, then Emacs uses the usual display conventions:

- Character codes 32 through 127 map to glyph codes 32 through 127.
- Codes 0 through 31 map to sequences of two glyphs, where the first glyph is the ASCII code for '`^`'.
- Character codes 128 through 255 map to sequences of four glyphs, where the first glyph is the ASCII code for '\', and the others represent digits.

The usual display conventions are also used for any character whose entry in the active display table is `nil`. This means that when you set up a display table, you need not specify explicitly what to do with each character, only the characters for which you want unusual behavior.

0.14.2 Glyphs

A glyph stands for an image that takes up a single character position on the screen. A glyph is represented in Lisp as an integer.

The meaning of each integer, as a glyph, is defined by the glyph table, which is the value of the variable `glyph-table`. It should be a vector; the gth element defines glyph code g. The possible definitions of a glyph code are:

integer Define this glyph code as an alias for code *integer*. This is used with X Windows to specify a face code.

string Send the characters in *string* to the terminal to output this glyph. This alternative is available only for character terminals, not with X.

`nil` This glyph is simple. On an ordinary terminal, the glyph code mod 256 is the character to output. With X, the glyph code mod 256 is character to output, and the glyph code divided by 256 specifies the *face code* to use while outputting it.

Any glyph code beyond the length of the glyph table is automatically simple.

If `glyph-table` is `nil`, then all possible glyph codes are simple.

A *face* is a named combination of a font and a pair of colors (foreground and background). A glyph code can specify a face id number to use for displaying that glyph.

0.14.3 ISO Latin 1

If you have a terminal that can handle the entire ISO Latin 1 character set, you can arrange to use that character set as follows:

```
(require 'disp-table)
(standard-display-8bit 0 255)
```

If you are editing buffers written in the ISO Latin 1 character set and your terminal doesn't handle anything but ASCII, you can load the file `iso-ascii` to set up a display table which makes the other ISO characters display as sequences of ASCII characters. For example, the character "o with umlaut" displays as '`{"o}`'.

Some European countries have terminals that don't support ISO Latin 1 but do support the special characters for that country's language. You can define a display table to work one language using such terminals. For an example, see `lisp/iso-swed.el`, which handles certain Swedish terminals.

You can load the appropriate display table for your terminal automatically by writing a terminal-specific Lisp file for the terminal type.

0.15 Overlays

You can use *overlays* to alter the appearance of a buffer's text on the screen. An overlay is an object which belongs to a particular buffer, and has a specified beginning and end. It also has properties which you can examine and set; these affect the display of the text within the overlay.

0.15.1 Overlay Properties

Overlay properties are like text properties in some respects, but the differences are more important than the similarities. Text properties are considered a part of the text; overlays are specifically considered not to be part of the text. Thus, copying text between various buffers and strings preserves text properties, but does not try to preserve overlays. Changing a buffer's text properties marks the buffer as modified, while moving an overlay or changing its properties does not.

`face` This property specifies a face for displaying the text within the overlay.

`priority` This property's value (which should be a nonnegative number) determines the priority of the overlay. The priority matters when two or more overlays cover the same character and both specify a face for display; the one whose `priority` value is larger takes priority over the other, and its face attributes override the face attributes of the lower priority overlay.

Currently, all overlays take priority over text properties. Please avoid using negative priority values, as we have not yet decided just what they should mean.

`window` If the `window` property is non-`nil`, then the overlay applies only on that window.

0.15.2 Overlay Functions

Use the functions `overlay-get` and `overlay-put` to access and set the properties of an overlay. They take arguments like `get` and `put`, except that the first argument is an overlay rather than a symbol.

To create an overlay, call `(make-overlay` *`start end`*`)`. You can specify the buffer as the third argument if you wish. To delete one, use `delete-overlay`.

Use `overlay-start`, `overlay-end` and `overlay-buffer` to examine the location and range of an overlay. Use `move-overlay` to change them; its arguments are *overlay*, *start*, *end* and (optionally) the buffer.

There are two functions to search for overlays: `overlays-at` and `next-overlay-change`. `overlays-at` returns a list of all the overlays containing a particular position. `(next-overlay-change` *`pos`*`)` returns the position of the next overlay beginning or end following *pos*.

0.16 Faces

A *face* is a named collection of graphical attributes: font, foreground color, background color and optional underlining. Faces control the display of text on the screen.

Each face has its own *face id number* which distinguishes faces at low levels within Emacs. However, for most purposes, you can refer to faces in Lisp programs by their names.

Each face name is meaningful for all frames, and by default it has the same meaning in all frames. But you can arrange to give a particular face name a special meaning in one frame if you wish.

0.16.1 Choosing a Face for Display

Here are all the ways to specify which face to use for display of text:

- With defaults. Each frame has a *default face*, whose id number is zero, which is used for all text that doesn't somehow specify another face.
- With text properties. A character may have a `face` property; if so, it's displayed with that face. If the character has a `mouse-face` property, that is used instead of the `face` property when the mouse is "near enough" to the character.
- With overlays. An overlay may have `face` and `mouse-face` properties too; they apply to all the text covered by the overlay.
- With special glyphs. Each glyph can specify a particular face id number.

If these various sources together specify more than one face for a particular character, Emacs merges the attributes of the various faces specified. The attributes of the faces of special glyphs come first; then come attributes of faces from overlays, followed by those from text properties, and last the default face.

When multiple overlays cover one character, an overlay with higher priority overrides those with lower priority.

If an attribute such as the font or a color is not specified in any of the above ways, the frame's own font or color is used.

See Section "Face Functions" in *The Emacs Lisp Reference Manual*, for functions to create and change faces.

0.17 New Input Event Formats

Mouse clicks, mouse movements and function keys no longer appear in the input stream as characters; instead, other kinds of Lisp objects represent them as input.

- An ordinary input character event consists of a *basic code* between 0 and 255, plus any or all of these *modifier bits*:

  meta — The 2**23 bit in the character code indicates a character typed with the meta key held down.

  control — The 2**22 bit in the character code indicates a non-ASCII control character.

  ASCII control characters such as `C-a` have special basic codes of their own, so Emacs needs no special bit to indicate them. Thus, the code for `C-a` is just 1.

  But if you type a control combination not in ASCII, such as `%` with the control key, the numeric value you get is the code for `%` plus 2**22 (assuming the terminal supports non-ASCII control characters).

  shift — The 2**21 bit in the character code indicates an ASCII control character typed with the shift key held down.

  For letters, the basic code indicates upper versus lower case; for digits and punctuation, the shift key selects an entirely different character with a different basic code. In order to keep within the ASCII character set whenever possible, Emacs avoids using the 2**21 bit for those characters.

However, ASCII provides no way to distinguish *C-A* from *C-a*, so Emacs uses the 2**21 bit in *C-A* and not in *C-a*.

hyper
: The 2**20 bit in the character code indicates a character typed with the hyper key held down.

super
: The 2**19 bit in the character code indicates a character typed with the super key held down.

alt
: The 2**18 bit in the character code indicates a character typed with the alt key held down. (On some terminals, the key labeled `ALT` is actually the meta key.)

In the future, Emacs may support a larger range of basic codes. We may also move the modifier bits to larger bit numbers. Therefore, you should avoid mentioning specific bit numbers in your program. Instead, the way to test the modifier bits of a character is with the function `event-modifiers` (see below).

- Function keys are represented as symbols. The symbol's name is the function key's label. For example, pressing a key labeled `F1` places the symbol `f1` in the input stream.

  There are a few exceptions to the symbol naming convention:

  `kp-add`, `kp-decimal`, `kp-divide`, ...
  : Keypad keys (to the right of the regular keyboard).

  `kp-0`, `kp-1`, ...
  : Keypad keys with digits.

  `kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`
  : Keypad PF keys.

  `left`, `up`, `right`, `down`
  : Cursor arrow keys

  You can use the modifier keys `CTRL`, `META`, `HYPER`, `SUPER`, `ALT` and `SHIFT` with function keys. The way to represent them is with prefixes in the symbol name:

  '`A-`'
  : The alt modifier.

  '`C-`'
  : The control modifier.

  '`H-`'
  : The hyper modifier.

  '`M-`'
  : The meta modifier.

  '`s-`'
  : The super modifier.

  '`S-`'
  : The shift modifier.

  Thus, the symbol for the key `F3` with `META` held down is *M-*`F3`. When you use more than one prefix, we recommend you write them in alphabetical order (though the order does not matter in arguments to the key-binding lookup and modification functions).

- Mouse events are represented as lists.

  If you press a mouse button and release it at the same location, this generates a "click" event. Mouse click events have this form:

  ```
  (button-symbol
  ```

```
(window (column . row)
 buffer-pos timestamp))
```

Here is what the elements normally mean:

button-symbol
: indicates which mouse button was used. It is one of the symbols `mouse-1`, `mouse-2`, ..., where the buttons are normally numbered left to right.

  You can also use prefixes '`A-`', '`C-`', '`H-`', '`M-`', '`S-`' and '`s-`' for modifiers alt, control, hyper, meta, shift and super, just as you would with function keys.

window
: is the window in which the click occurred.

column
row
: are the column and row of the click, relative to the top left corner of *window*, which is `(0 . 0)`.

buffer-pos
: is the buffer position of the character clicked on.

timestamp
: is the time at which the event occurred, in milliseconds. (Since this value wraps around the entire range of Emacs Lisp integers in about five hours, it is useful only for relating the times of nearby events.)

The meanings of *buffer-pos*, *row* and *column* are somewhat different when the event location is in a special part of the screen, such as the mode line or a scroll bar.

If the position is in the window's scroll bar, then *buffer-pos* is the symbol `vertical-scroll-bar`, and the pair `(column . row)` is replaced with a pair `(portion . whole)`, where *portion* is the distance of the click from the top or left end of the scroll bar, and *whole* is the length of the entire scroll bar.

If the position is on a mode line or the vertical line separating *window* from its neighbor to the right, then *buffer-pos* is the symbol `mode-line` or `vertical-line`. In this case *row* and *column* do not have meaningful data.

- Releasing a mouse button above a different character position generates a "drag" event, which looks like this:

```
(button-symbol
 (window1 (column1 . row1)
  buffer-pos1 timestamp1)
 (window2 (column2 . row2)
  buffer-pos2 timestamp2))
```

The name of *button-symbol* contains the prefix '`drag-`'. The second and third elements of the event give the starting and ending position of the drag.

The '`drag-`' prefix follows the modifier key prefixes such as '`C-`' and '`M-`'.

If `read-key-sequence` receives a drag event which has no key binding, and the corresponding click event does have a binding, it changes the drag event into a click event at the drag's starting position. This means that you don't have to distinguish between click and drag events unless you want to.

- Click and drag events happen when you release a mouse button. Another kind of event happens when you press a button. It looks just like a click event, except that the name of *button-symbol* contains the prefix '`down-`'. The '`down-`' prefix follows the modifier key prefixes such as '`C-`' and '`M-`'.

  The function `read-key-sequence`, and the Emacs command loop, ignore any down events that don't have command bindings. This means that you need not worry about defining down events unless you want them to do something. The usual reason to define a down event is so that you can track mouse motion until the button is released.

- For example, if the user presses and releases the left mouse button over the same location, Emacs generates a sequence of events like this:

```
(down-mouse-1 (#<window 18 on NEWS> 2613 (0 . 38) -864320))
(mouse-1      (#<window 18 on NEWS> 2613 (0 . 38) -864180))
```

  Or, while holding the control key down, the user might hold down the second mouse button, and drag the mouse from one line to the next. That produces two events, as shown here:

```
(C-down-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219))
(C-drag-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219)
                (#<window 18 on NEWS> 3510 (0 . 28) -729648))
```

  Or, while holding down the meta and shift keys, the user might press the second mouse button on the window's mode line, and then drag the mouse into another window. That produces an event like this:

```
(M-S-down-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844))
(M-S-drag-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844)
                  (#<window 20 on carlton-sanskrit.tex> 161 (33 . 3)
                   -453816))
```

- A key sequence that starts with a mouse click is read using the keymaps of the buffer in the window clicked on, not the current buffer.

  This does not imply that clicking in a window selects that window or its buffer. The execution of the command begins with no change in the selected window or current buffer. However, the command can switch windows or buffers if programmed to do so.

- Mouse motion events are represented by lists. During the execution of the body of a `track-mouse` form, moving the mouse generates events that look like this:

```
(mouse-movement (window (column . row)
                 buffer-pos timestamp))
```

  The second element of the list describes the current position of the mouse, just as in a mouse click event.

  Outside of `track-mouse` forms, Emacs does not generate events for mere motion of the mouse, and these events do not appear.

- Focus shifts between frames are represented by lists.

  When the mouse shifts temporary input focus from one frame to another, Emacs generates an event like this:

```
(switch-frame new-frame)
```

  where *new-frame* is the frame switched to.

  In X windows, most window managers are set up so that just moving the mouse into a window is enough to set the focus there. As far as the user is concerned, Emacs behaves

consistently with this. However, there is no need for the Lisp program to know about the focus change until some other kind of input arrives. So Emacs generates the focus event only when the user actually types a keyboard key or presses a mouse button in the new frame; just moving the mouse between frames does not generate a focus event.

The global key map usually binds this event to the `internal-select-frame` function, so that characters typed at a frame apply to that frame's selected window.

If the user switches frames in the middle of a key sequence, then Emacs delays the `switch-frame` event until the key sequence is over. For example, suppose *C-c C-a* is a key sequence in the current buffer's keymaps. If the user types *C-c*, moves the mouse to another frame, and then types *C-a*, `read-key-sequence` returns the sequence `"\C-c\C-a"`, and the next call to `read-event` or `read-key-sequence` will return the `switch-frame` event.

0.18 Working with Input Events

- Functions which work with key sequences now handle non-character events. Functions like `define-key`, `global-set-key`, and `local-set-key` used to accept strings representing key sequences; now, since events may be arbitrary lisp objects, they also accept vectors. The function `read-key-sequence` may return a string or a vector, depending on whether or not the sequence read contains only characters.

  List events may be represented by the symbols at their head; to bind clicks of the left mouse button, you need only present the symbol `mouse-1`, not an entire mouse click event. If you do put an event which is a list in a key sequence, only the event's head symbol is used in key lookups.

  For example, to globally bind the left mouse button to the function `mouse-set-point`, you could evaluate this:

  ```
  (global-set-key [mouse-1] 'mouse-set-point)
  ```

  To bind the sequence *C-c* `F1` to the command `tex-view` in `tex-mode-map`, you could evaluate this:

  ```
  (define-key tex-mode-map [?\C-c f1] 'tex-view)
  ```

  To find the binding for the function key labeled `NEXT` in `minibuffer-local-map`, you could evaluate this:

  ```
  (lookup-key minibuffer-local-map [next])
       ⇒ next-history-element
  ```

  If you call the function `read-key-sequence` and then press *C-x C-*`F5`, here is how it behaves:

  ```
  (read-key-sequence "Press `C-x C-F5': ")
       ⇒ [24 C-f5]
  ```

  Note that '`24`' is the character *C-x*.

- The documentation functions (`single-key-description`, `key-description`, etc.) now handle the new event types. Wherever a string of keyboard input characters was acceptable in previous versions of Emacs, a vector of events should now work.

- Special parts of a window can have their own bindings for mouse events.

  When mouse events occur in special parts of a window, such as a mode line or a scroll bar, the event itself shows nothing special—only the symbol that would normally

represent that mouse button and modifier keys. The information about the screen region is kept in other parts of the event list. But `read-key-sequence` translates this information into imaginary prefix keys, all of which are symbols: `mode-line`, `vertical-line`, and `vertical-scroll-bar`.

For example, if you call `read-key-sequence` and then click the mouse on the window's mode line, this is what happens:

```
(read-key-sequence "Click on the mode line: ")
     ⇒ [mode-line (mouse-1 (#<window 6 on NEWS> mode-line
                            (40 . 63) 5959987))]
```

You can define meanings for mouse clicks in special window regions by defining key sequences using these imaginary prefix keys. For example, here is how to bind the third mouse button on a window's mode line delete the window:

```
(global-set-key [mode-line mouse-3] 'mouse-delete-window)
```

Here's how to bind the middle button (modified by `META`) on the vertical line at the right of a window to scroll the window to the left.

```
(global-set-key [vertical-line M-mouse-2] 'scroll-left)
```

- Decomposing an event symbol.

  Each symbol used to identify a function key or mouse button has a property named `event-symbol-elements`, which is a list containing an unmodified version of the symbol, followed by modifiers the symbol name contains. The modifiers are symbols; they include `shift`, `control`, and `meta`. In addition, a mouse event symbol has one of `click`, `drag`, and `down`. For example:

```
(get 'f5 'event-symbol-elements)
     ⇒ (f5)
(get 'C-f5 'event-symbol-elements)
     ⇒ (f5 control)
(get 'M-S-f5 'event-symbol-elements)
     ⇒ (f5 meta shift)
(get 'mouse-1 'event-symbol-elements)
     ⇒ (mouse-1 click)
(get 'down-mouse-1 'event-symbol-elements)
     ⇒ (mouse-1 down)
```

  Note that the `event-symbol-elements` property for a mouse click explicitly contains `click`, but the event symbol name itself does not contain '`click`'.

- Use `read-event` to read input if you want to accept any kind of event. The old function `read-char` now discards events other than keyboard characters.
- `last-command-char` and `last-input-char` can now hold any kind of event.
- The new variable `unread-command-events` is much like `unread-command-char`. Its value is a list of events of any type, to be processed as command input in order of appearance in the list.
- The function `this-command-keys` may return a string or a vector, depending on whether or not the sequence read contains only characters. You may need to upgrade code which uses this function.

  The function `recent-keys` now returns a vector of events. You may need to upgrade code which uses this function.

- A keyboard macro's definition can now be either a string or a vector. All that really matters is what elements it has. If the elements are all characters, then the macro can be a string; otherwise, it has to be a vector.
- The variable `last-event-frame` records which frame the last input event was directed to. Usually this is the frame that was selected when the event was generated, but if that frame has redirected input focus to another frame, `last-event-frame` is the frame to which the event was redirected.
- The interactive specification now allows a new code letter '`e`' to simplify commands bound to events which are lists. This code supplies as an argument the complete event object.

  You can use '`e`' more than once in a single command's interactive specification. If the key sequence which invoked the command has *n* events with parameters, the *n*th '`e`' provides the *n*th parameterized event. Events which are not lists, such as function keys and ASCII keystrokes, do not count where '`e`' is concerned.
- You can extract the starting and ending position values from a mouse button or motion event using the two functions `event-start` and `event-end`. These two functions return different values for drag and motion events; for click and button-down events, they both return the position of the event.
- The position, a returned by `event-start` and `event-end`, is a list of this form:

  ```
  (window buffer-position (col . row) timestamp)
  ```

  You can extract parts of this list with the functions `posn-window`, `posn-point`, `posn-col-row`, and `posn-timestamp`.
- The function `scroll-bar-scale` is useful for computing where to scroll to in response to a mouse button event from a scroll bar. It takes two arguments, *ratio* and *total*, and in effect multiplies them. We say "in effect" because *ratio* is not a number; rather a pair `(num . denom)`.

  Here's the usual way to use `scroll-bar-scale`:

  ```
  (scroll-bar-scale (posn-col-row (event-start event))
                    (buffer-size))
  ```

0.19 Putting Keyboard Events in Strings

In most of the places where strings are used, we conceptualize the string as containing text characters—the same kind of characters found in buffers or files. Occasionally Lisp programs use strings which conceptually contain keyboard characters; for example, they may be key sequences or keyboard macro definitions. There are special rules for how to put keyboard characters into a string, because they are not limited to the range of 0 to 255 as text characters are.

A keyboard character typed using the `META` key is called a *meta character*. The numeric code for such an event includes the 2**23 bit; it does not even come close to fitting in a string. However, earlier Emacs versions used a different representation for these characters, which gave them codes in the range of 128 to 255. That did fit in a string, and many Lisp programs contain string constants that use '`\M-`' to express meta characters, especially as the argument to `define-key` and similar functions.

We provide backward compatibility to run those programs with special rules for how to put a keyboard character event in a string. Here are the rules:

- If the keyboard event value is in the range of 0 to 127, it can go in the string unchanged.
- The meta variants of those events, with codes in the range of 2**23 to 2**23+127, can also go in the string, but you must change their numeric values. You must set the 2**7 bit instead of the 2**23 bit, resulting in a value between 128 and 255.
- Other keyboard character events cannot fit in a string. This includes keyboard events in the range of 128 to 255.

Functions such as `read-key-sequence` that can construct strings containing events follow these rules.

When you use the read syntax '`\M-`' in a string, it produces a code in the range of 128 to 255—the same code that you get if you modify the corresponding keyboard event to put it in the string. Thus, meta events in strings work consistently regardless of how they get into the strings.

New programs can avoid dealing with these rules by using vectors instead of strings for key sequences when there is any possibility that these issues might arise.

The reason we changed the representation of meta characters as keyboard events is to make room for basic character codes beyond 127, and support meta variants of such larger character codes.

0.20 Menus

You can now define menus conveniently as keymaps. Menus are normally used with the mouse, but they can work with the keyboard also.

0.20.1 Defining Menus

A keymap is suitable for menu use if it has an *overall prompt string*, which is a string that appears as an element of the keymap. It should describes the purpose of the menu. The easiest way to construct a keymap with a prompt string is to specify the string as an argument when you run `make-keymap` or `make-sparse-keymap`.

The individual bindings in the menu keymap should also have prompt strings; these strings are the items in the menu. A binding with a prompt string looks like this:

```
(char string . real-binding)
```

As far as `define-key` is concerned, the string is part of the character's binding—the binding looks like this:

```
(string . real-binding).
```

However, only *real-binding* is used for executing the key.

You can also supply a second string, called the help string, as follows:

```
(char string help-string . real-binding)
```

Currently Emacs does not actually use *help-string*; it knows only how to ignore *help-string* in order to extract *real-binding*. In the future we hope to make *help-string* serve as longer documentation for the menu item, available on request.

The prompt string for a binding should be short—one or two words. Its meaning should describe the command it corresponds to.

If *real-binding* is `nil`, then *string* appears in the menu but cannot be selected.

If *real-binding* is a symbol, and has a non-`nil` `menu-enable` property, that property is an expression which controls whether the menu item is enabled. Every time the keymap is used to display a menu, Emacs evaluates the expression, and it enables the menu item only if the expression's value is non-`nil`. When a menu item is disabled, it is displayed in a "fuzzy" fashion, and cannot be selected with the mouse.

0.20.2 Menus and the Mouse

The way to make a menu keymap produce a menu is to make it the definition of a prefix key.

When the prefix key ends with a mouse event, Emacs handles the menu keymap by popping up a visible menu that you can select from with the mouse. When you click on a menu item, the event generated is whatever character or symbol has the binding which brought about that menu item.

A single keymap can appear as multiple panes, if you explicitly arrange for this. The way to do this is to make a keymap for each pane, then create a binding for each of those maps in the main keymap of the menu. Give each of these bindings a prompt string that starts with '`@`'. The rest of the prompt string becomes the name of the pane. See the file `lisp/mouse.el` for an example of this. Any ordinary bindings with prompt strings are grouped into one pane, which appears along with the other panes explicitly created for the submaps.

You can also get multiple panes from separate keymaps. The full definition of a prefix key always comes from merging the definitions supplied by the various active keymaps (minor modes, local, and global). When more than one of these keymaps is a menu, each of them makes a separate pane or panes.

0.20.3 Menus and the Keyboard

When a prefix key ending with a keyboard event (a character or function key) has a definition that is a menu keymap, you can use the keyboard to choose a menu item.

Emacs displays the menu alternatives in the echo area. If they don't all fit at once, type `SPC` to see the next line of alternatives. If you keep typing `SPC`, you eventually get to the end of the menu and then cycle around to the beginning again.

When you have found the alternative you want, type the corresponding character—the one whose binding is that alternative.

In a menu intended for keyboard use, each menu item must clearly indicate what character to type. The best convention to use is to make the character the first letter of the menu item prompt string. That is something users will understand without being told.

0.20.4 The Menu Bar

Under X Windows, each frame can have a *menu bar*—a permanently displayed menu stretching horizontally across the top of the frame. The items of the menu bar are the subcommands of the fake "function key" `menu-bar`, as defined by all the active keymaps.

To add an item to the menu bar, invent a fake "function key" of your own (let's call it *key*), and make a binding for the key sequence `[menu-bar` *key*`]`. Most often, the binding is a menu keymap, so that pressing a button on the menu bar item leads to another menu.

In order for a frame to display a menu bar, its `menu-bar-lines` property must be greater than zero. Emacs uses just one line for the menu bar itself; if you specify more than one line, the other lines serve to separate the menu bar from the windows in the frame. We recommend you try one or two as the `menu-bar-lines` value.

0.21 Keymaps

- The representation of keymaps has changed to support the new event types. All keymaps now have the form `(keymap` *`element element ...`*`)`. Each *element* takes one of the following forms:

  prompt-string
  : A string as an element of the keymap marks the keymap as a menu, and serves as the overall prompt string for it.

  `(`*`key . binding`*`)`
  : A cons cell binds *key* to *definition*. Here *key* may be any sort of event head—a character, a function key symbol, or a mouse button symbol.

  vector
  : A vector of 128 elements binds all the ASCII characters; the *n*th element holds the binding for character number *n*.

  `(t .` *`binding`*`)`
  : A cons cell whose CAR is `t` is a default binding; anything not bound by previous keymap elements is given *binding* as its binding.

    Default bindings are important because they allow a keymap to bind all possible events without having to enumerate all the possible function keys and mouse clicks, with all possible modifier prefixes.

    The function `lookup-key` (and likewise other functions for examining a key binding) normally report only explicit bindings of the specified key sequence; if there is none, they return `nil`, even if there is a default binding that would apply to that key sequence if it were actually typed in. However, these functions now take an optional argument *accept-defaults* which, if non-`nil`, says to consider default bindings.

    Note that if a vector in the keymap binds an ASCII character to `nil` (thus making it "unbound"), the default binding does not apply to the character. Think of the vector element as an explicit binding of `nil`.

    Note also that if the keymap for a minor or major mode contains a default binding, it completely masks out any lower-priority keymaps.

- A keymap can now inherit from another keymap. To do this, make the latter keymap the "tail" of the new one. Such a keymap looks like this:

  ```
  (keymap bindings... . other-keymap)
  ```

  The effect is that this keymap inherits all the bindings of *other-keymap*, but can add to them or override them with *bindings*. Subsequent changes in the bindings of *other-keymap do* affect this keymap.

  For example,

  ```
  (setq my-mode-map (cons 'keymap text-mode-map))
  ```

makes a keymap that by default inherits all the bindings of Text mode—whatever they may be at the time a key is looked up. Any bindings made explicitly in `my-mode-map` override the bindings inherited from Text mode, however.

- Minor modes can now have local keymaps. Thus, a key can act a special way when a minor mode is in effect, and then revert to the major mode or global definition when the minor mode is no longer in effect. The precedence of keymaps is now: minor modes (in no particular order), then major mode, and lastly the global map.

  The new `current-minor-mode-maps` function returns a list of all the keymaps of currently enabled minor modes, in the other that they apply.

  To set up a keymap for a minor mode, add an element to the alist `minor-mode-map-alist`. Its elements look like this:

  ```
  (symbol . keymap)
  ```

  The keymap *keymap* is active whenever *symbol* has a non-`nil` value. Use for *symbol* the variable which indicates whether the minor mode is enabled.

  When more than one minor mode keymap is active, their order of precedence is the order of `minor-mode-map-alist`. But you should design minor modes so that they don't interfere with each other, and if you do this properly, the order will not matter.

  The function `minor-mode-key-binding` returns a list of all the active minor mode bindings of *key*. More precisely, it returns an alist of pairs `(modename . binding)`, where *modename* is the the variable which enables the minor mode, and *binding* is *key*'s definition in that mode. If *key* has no minor-mode bindings, the value is `nil`.

  If the first binding is a non-prefix, all subsequent bindings from other minor modes are omitted, since they would be completely shadowed. Similarly, the list omits non-prefix bindings that follow prefix bindings.

- The new function `copy-keymap` copies a keymap, producing a new keymap with the same key bindings in it. If the keymap contains other keymaps directly, these subkeymaps are copied recursively.

  If you want to, you can define a prefix key with a binding that is a symbol whose function definition is another keymap. In this case, `copy-keymap` does not look past the symbol; it doesn't copy the keymap inside the symbol.

- `substitute-key-definition` now accepts an optional fourth argument, which is a keymap to use as a template.

  ```
  (substitute-key-definition olddef newdef keymap oldmap)
  ```

  finds all characters defined in *oldmap* as *olddef*, and defines them in *keymap* as *newdef*.

  In addition, this function now operates recursively on the keymaps that define prefix keys within *keymap* and *oldmap*.

0.22 Minibuffer Features

The minibuffer input functions `read-from-minibuffer` and `completing-read` have new features.

0.22.1 Minibuffer History

A new optional argument *hist* specifies which history list to use. If you specify a variable (a symbol), that variable is the history list. If you specify a cons cell `(variable . startpos)`,

then *variable* is the history list variable, and *startpos* specifies the initial history position (an integer, counting from zero which specifies the most recent element of the history).

If you specify *startpos*, then you should also specify that element of the history as *initial-input*, for consistency.

If you don't specify *hist*, then the default history list `minibuffer-history` is used. Other standard history lists that you can use when appropriate include `query-replace-history`, `command-history`, and `file-name-history`.

The value of the history list variable is a list of strings, most recent first. You should set a history list variable to `nil` before using it for the first time.

`read-from-minibuffer` and `completing-read` add new elements to the history list automatically, and provide commands to allow the user to reuse items on the list. The only thing your program needs to do to use a history list is to initialize it and to pass its name to the input functions when you wish. But it is safe to modify the list by hand when the minibuffer input functions are not using it.

0.22.2 Other Minibuffer Features

The *initial* argument to `read-from-minibuffer` and other minibuffer input functions can now be a cons cell (*`string . position`*). This means to start off with *string* in the minibuffer, but put the cursor *position* characters from the beginning, rather than at the end.

In `read-no-blanks-input`, the *initial* argument is now optional; if it is omitted, the initial input string is the empty string.

0.23 New Features for Defining Commands

- If the interactive specification begins with '`@`', this means to select the window under the mouse. This selection takes place before doing anything else with the command.

  You can use both '`@`' and '`*`' together in one command; they are processed in order of appearance.

- Prompts in an interactive specification can incorporate the values of the preceding arguments. Emacs replaces '`%`'-sequences (as used with the `format` function) in the prompt with the interactive arguments that have been read so far. For example, a command with this interactive specification

  ```
  (interactive "sReplace: \nsReplace %s with: ")
  ```

  prompts for the first argument with '`Replace: `', and then prompts for the second argument with '`Replace` *`foo`* `with: `', where *foo* is the string read as the first argument.

- If a command name has a property `enable-recursive-minibuffers` which is non-`nil`, then the command can use the minibuffer to read arguments even if it is invoked from the minibuffer. The minibuffer command `next-matching-history-element` (normally bound to *`M-s`* in the minibuffer) uses this feature.

0.24 New Features for Reading Input

- The function `set-input-mode` now takes four arguments. The last argument is optional. Their names are *interrupt*, *flow*, *meta* and *quit*.

The argument *interrupt* says whether to use interrupt-driven input. Non-`nil` means yes, and `nil` means no (use CBREAK mode).

The argument *flow* says whether to enable terminal flow control. Non-`nil` means yes.

The argument *meta* controls support for input character codes above 127. If *meta* is `t`, Emacs converts characters with the 8th bit set into Meta characters. If *meta* is `nil`, Emacs disregards the 8th bit; this is necessary when the terminal uses it as a parity bit. If *meta* is neither `t` nor `nil`, Emacs uses all 8 bits of input unchanged. This is good for terminals using European 8-bit character sets.

If *quit* non-`nil`, it is the character to use for quitting. (Normally this is *C-g*.)

- The variable `meta-flag` has been deleted; use `set-input-mode` to enable or disable support for a `META` key. This change was made because `set-input-mode` can send the terminal the appropriate commands to enable or disable operation of the `META` key.
- The new variable `extra-keyboard-modifiers` lets Lisp programs "press" the modifier keys on the keyboard. The value is a bit mask:

  1 The `SHIFT` key.

  2 The `LOCK` key.

  4 The `CTL` key.

  8 The `META` key.

  When you use X windows, the program can press any of the modifier keys in this way. Otherwise, only the `CTL` and `META` keys can be virtually pressed.
- You can use the new function `keyboard-translate` to set up `keyboard-translate-table` conveniently.
- Y-or-n questions using the `y-or-n-p` function now accept *C-]* (usually mapped to `abort-recursive-edit`) as well as *C-g* to quit.
- The variable `num-input-keys` is the total number of key sequences that the user has typed during this Emacs session.
- A new Lisp variable, `function-key-map`, holds a keymap which describes the character sequences sent by function keys on an ordinary character terminal. This uses the same keymap data structure that is used to hold bindings of key sequences, but it has a different meaning: it specifies translations to make while reading a key sequence.

  If `function-key-map` "binds" a key sequence *k* to a vector *v*, then when *k* appears as a subsequence *anywhere* in a key sequence, it is replaced with *v*.

  For example, VT100 terminals send `ESC` *O P* when the "keypad" PF1 key is pressed. Thus, on a VT100, `function-key-map` should "bind" that sequence to `[pf1]`. This specifies translation of `ESC` *O P* into `PF1` anywhere in a key sequence.

  Thus, typing *C-c* `PF1` sends the character sequence *C-c* `ESC` *O P*, but `read-key-sequence` translates this back into *C-c* `PF1`, which it returns as the vector `[?\C-c PF1]`.

  Entries in `function-key-map` are ignored if they conflict with bindings made in the minor mode, local, or global keymaps.

  The value of `function-key-map` is usually set up automatically according to the terminal's Terminfo or Termcap entry, and the terminal-specific Lisp files. Emacs comes

with a number of terminal-specific files for many common terminals; their main purpose is to make entries in `function-key-map` beyond those that can be deduced from Termcap and Terminfo.

- The variable `key-translation-map` works like `function-key-map` except for two things:
  - `key-translation-map` goes to work after `function-key-map` is finished; it receives the results of translation by `function-key-map`.
  - `key-translation-map` overrides actual key bindings.

  The intent of `key-translation-map` is for users to map one character set to another, including ordinary characters normally bound to `self-insert-command`.

0.25 New Syntax Table Features

- You can use two new functions to move across characters in certain syntax classes.

  `skip-syntax-forward` moves point forward across characters whose syntax classes are mentioned in its first argument, a string. It stops when it encounters the end of the buffer, or position *lim* (the optional second argument), or a character it is not supposed to skip. The function `skip-syntax-backward` is similar but moves backward.
- The new function `forward-comment` moves point by comments. It takes one argument, *count*; it moves point forward across *count* comments (backward, if *count* is negative). If it finds anything other than a comment or whitespace, it stops, leaving point at the far side of the last comment found. It also stops after satisfying *count*.
- The new variable `words-include-escapes` affects the behavior of `forward-word` and everything that uses it. If it is non-`nil`, then characters in the "escape" and "character quote" syntax classes count as part of words.
- There are two new syntax flags for use in syntax tables.
  - The prefix flag.

    The '`p`' flag identifies additional "prefix characters" in Lisp syntax. You can set this flag with `modify-syntax-entry` by including the letter '`p`' in the syntax specification.

    These characters are treated as whitespace when they appear between expressions. When they appear withing an expression, they are handled according to their usual syntax codes.

    The function `backward-prefix-chars` moves back over these characters, as well as over characters whose primary syntax class is prefix ('`'`').
  - The '`b`' comment style flag.

    Emacs can now supports two comment styles simultaneously. (This is for the sake of C++.) More specifically, it can recognize two different comment-start sequences. Both must share the same first character; only the second character may differ. Mark the second character of the '`b`'-style comment start sequence with the '`b`' flag. You can set this flag with `modify-syntax-entry` by including the letter '`b`' in the syntax specification.

    The two styles of comment can have different comment-end sequences. A comment-end sequence (one or two characters) applies to the '`b`' style if its first character has the '`b`' flag set; otherwise, it applies to the '`a`' style.

The appropriate comment syntax settings for C++ are as follows:

‘/’ ‘124b’

‘*’ ‘23’

newline ‘>b’

Thus ‘/*’ is a comment-start sequence for ‘a’ style, ‘//’ is a comment-start sequence for ‘b’ style, ‘*/’ is a comment-end sequence for ‘a’ style, and newline is a comment-end sequence for ‘b’ style.

0.26 The Case Table

You can customize case conversion using the new case table feature. A case table is a collection of strings that specifies the mapping between upper case and lower case letters. Each buffer has its own case table. You need a case table if you are using a language which has letters that are not standard ASCII letters.

A case table is a list of this form:

```
(downcase upcase canonicalize equivalences)
```

where each element is either `nil` or a string of length 256. The element *downcase* says how to map each character to its lower-case equivalent. The element *upcase* maps each character to its upper-case equivalent. If lower and upper case characters are in 1-1 correspondence, use `nil` for *upcase*; then Emacs deduces the upcase table from *downcase*.

For some languages, upper and lower case letters are not in 1-1 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 directions.

The element *canonicalize* maps each character to a canonical equivalent; any two characters that are related by case-conversion have the same canonical equivalent character.

The element *equivalences* is a map that cyclicly permutes each equivalence class (of characters with the same canonical equivalent).

You can provide `nil` for both *canonicalize* and *equivalences*, in which case both are deduced from *downcase* and *upcase*.

Here are the functions for working with case tables:

`case-table-p` is a predicate that says whether a Lisp object is a valid case table.

`set-standard-case-table` takes one argument and makes that argument the case table for new buffers created subsequently. `standard-case-table` returns the current value of the new buffer case table.

`current-case-table` returns the case table of the current buffer. `set-case-table` sets the current buffer's case table to the argument.

`set-case-syntax-pair` is a convenient function for specifying a pair of letters, upper case and lower case. Call it with two arguments, the upper case letter and the lower case letter. It modifies the standard case table and a few syntax tables that are predefined in Emacs. This function is intended as a subroutine for packages that define non-ASCII character sets.

Load the library `iso-syntax` to set up the syntax and case table for the 256 bit ISO Latin 1 character set.

0.27 New Features for Dealing with Buffers

- The new function `buffer-modified-tick` returns a buffer's modification-count that ticks every time the buffer is modified. It takes one optional argument, which is the buffer you want to examine. If the argument is `nil` (or omitted), the current buffer is used.
- `buffer-disable-undo` is a new name for the function formerly known as `buffer-flush-undo`. This turns off recording of undo information in the buffer given as argument.
- The new function `generate-new-buffer-name` chooses a name that would be unique for a new buffer—but does not create the buffer. Give it one argument, a starting name. It produces a name not in use for a buffer by appending a number inside of '`<...>`'.
- The function `rename-buffer` now takes an optional second argument which tells it that if the specified new name corresponds to an existing buffer, it should use `generate-new-buffer-name` to modify the name to be unique, rather than signaling an error.

  `rename-buffer` now returns the name to which the buffer was renamed.
- The function `list-buffers` now looks at the local variable `list-buffers-directory` in each non-file-visiting buffer, and shows its value where the file would normally go. Dired sets this variable in each Dired buffer, so the buffer list now shows which directory each Dired buffer is editing.
- The function `other-buffer` now takes an optional second argument *visible-ok* which, if non-`nil`, indicates that buffers currently being displayed in windows may be returned even if there are other buffers not visible. Normally, `other-buffer` returns a currently visible buffer only as a last resort, if there are no suitable nonvisible buffers.
- The hook `kill-buffer-hook` now runs whenever a buffer is killed.

0.28 Local Variables Features

- If a local variable name has a non-`nil` `permanent-local` property, then `kill-all-local-variables` does not kill it. Such local variables are "permanent"—they remain unchanged even if you select a different major mode.

  Permanent locals are useful when they have to do with where the file came from or how to save it, rather than with how to edit the contents.
- The function `make-local-variable` now never changes the value of the variable that it makes local. If the variable had no value before, it still has no value after becoming local.
- The new function `default-boundp` tells you whether a variable has a default value (as opposed to being unbound in its default value). If `(default-boundp 'foo)` returns `nil`, then `(default-value 'foo)` would get an error.

  `default-boundp` is to `default-value` as `boundp` is to `symbol-value`.
- The special forms `defconst` and `defvar`, when the variable is local in the current buffer, now set the variable's default value rather than its local value.

0.29 New Features for Subprocesses

- `call-process` and `call-process-region` now return a value that indicates how the synchronous subprocess terminated. It is either a number, which is the exit status of a process, or a signal name represented as a string.
- `process-status` now returns `open` and `closed` as the status values for network connections.
- The standard asynchronous subprocess features work on VMS now, and the special VMS asynchronous subprocess functions have been deleted.
- You can use the transaction queue feature for more convenient communication with subprocesses using transactions.

  Call `tq-create` to create a transaction queue communicating with a specified process. Then you can call `tq-enqueue` to send a transaction. `tq-enqueue` takes these five arguments:

  ```
  (tq-enqueue tq question regexp closure fn)
  ```

  tq is the queue to use. (Specifying the queue has the effect of specifying the process to talk to.) The argument *question* is the outgoing message which starts the transaction. The argument *fn* is the function to call when the corresponding answer comes back; it is called with two arguments: *closure*, and the answer received.

  The argument *regexp* is a regular expression to match the entire answer; that's how `tq-enqueue` tells where the answer ends.

  Call `tq-close` to shut down a transaction queue and terminate its subprocess.
- The function `signal-process` sends a signal to process *pid*, which need not be a child of Emacs. The second argument *signal* specifies which signal to send; it should be an integer.

0.30 New Features for Dealing with Times And Time Delays

- The new function `current-time` returns the system's time value as a list of three integers: `(high low microsec)`. The integers *high* and *low* combine to give the number of seconds since 0:00 January 1, 1970, which is *high* * 2**16 + *low*.

  microsec gives the microseconds since the start of the current second (or 0 for systems that return time only on the resolution of a second).
- The function `current-time-string` accepts an optional argument *time-value*. If given, this specifies a time to format instead of the current time. The argument should be a cons cell containing two integers, or a list whose first two elements are integers. Thus, you can use times obtained from `current-time` (see above) and from `file-attributes`.
- You can now find out the user's time zone using `current-time-zone`.

  The value has the form `(OFFSET name)`. Here *offset* is an integer giving the number of seconds ahead of UTC (east of Greenwich). A negative value means west of Greenwich. The second element, *name* is a string giving the name of the time zone. Both elements change when daylight savings time begins or ends; if the user has specified a time zone that does not use a seasonal time adjustment, then the value is constant through time.

If the operating system doesn't supply all the information necessary to compute the value, both elements of the list are `nil`.

The optional argument *time-value*, if given, specifies a time to analyze instead of the current time. The argument should be a cons cell containing two integers, or a list whose first two elements are integers. Thus, you can use times obtained from `current-time` and from `file-attributes`.

- `sit-for`, `sleep-for` now let you specify the time period in milliseconds as well as in seconds. The first argument gives the number of seconds, as before, and the optional second argument gives additional milliseconds. The time periods specified by these two arguments are added together.

  Not all systems support this; you get an error if you specify nonzero milliseconds and it isn't supported.

  `sit-for` also accepts an optional third argument *nodisp*. If this is non-`nil`, `sit-for` does not redisplay. It still waits for the specified time or until input is available.

- `accept-process-output` now accepts a timeout specified by optional second and third arguments. The second argument specifies the number of seconds, while the third specifies the number of milliseconds. The time periods specified by these two arguments are added together.

  Not all systems support this; you get an error if you specify nonzero milliseconds and it isn't supported.

  The function returns `nil` if the timeout expired before output arrived, or non-`nil` if it did get some output.

- You can set up a timer to call a function at a specified future time. To do so, call `run-at-time`, like this:

  ```
  (run-at-time time repeat function args...)
  ```

  Here, *time* is a string saying when to call the function. The argument *function* is the function to call later, and *args* are the arguments to give it when it is called.

  The argument *repeat* specifies how often to repeat the call. If *repeat* is `nil`, there are no repetitions; *function* is called just once, at *time*. If *repeat* is an integer, it specifies a repetition period measured in seconds.

  Absolute times may be specified in a wide variety of formats; The form '*hour*`:`*min*`:`*sec timezone month/day/year*', where all fields are numbers, works; the format that `current-time-string` returns is also allowed.

  To specify a relative time, use numbers followed by units. For example:

  '`1 min`' denotes 1 minute from now.

  '`1 min 5 sec`'
  denotes 65 seconds from now.

  '`1 min 2 sec 3 hour 4 day 5 week 6 fortnight 7 month 8 year`'
  denotes exactly 103 months, 123 days, and 10862 seconds from now.

  If *time* is an integer, that specifies a relative time measured in seconds.

To cancel the requested future action, pass the value that `run-at-time` returned to the function `cancel-timer`.

0.31 Profiling Lisp Programs

You can now make execution-time profiles of Emacs Lisp programs using the `profile` library. See the file `profile.el` for instructions; if you have written a Lisp program big enough to be worth profiling, you can surely understand them.

0.32 New Features for Lisp Debuggers

- You can now specify which kinds of errors should invoke the Lisp debugger by setting the variable `debug-on-error` to a list of error conditions. For example, if you set it to the list `(void-variable)`, then only errors about a variable that has no value invoke the debugger.
- The variable `command-debug-status` is used by Lisp debuggers. It records the debugging status of current interactive command. Each time a command is called interactively, this variable is bound to `nil`. The debugger can set this variable to leave information for future debugger invocations during the same command.

  The advantage of this variable over some other variable in the debugger itself is that the data will not be visible for any other command invocation.
- The function `backtrace-frame` is intended for use in Lisp debuggers. It returns information about what a frame on the Lisp call stack is doing. You specify one argument, which is the number of stack frames to count up from the current execution point.

  If that stack frame has not evaluated the arguments yet (or is a special form), the value is `(nil` *`function arg-forms...`*`)`.

  If that stack frame has evaluated its arguments and called its function already, the value is `(t` *`function arg-values...`*`)`.

  In the return value, *function* is whatever was supplied as CAR of evaluated list, or a `lambda` expression in the case of a macro call. If the function has a `&rest` argument, that is represented as the tail of the list *arg-values*.

  If the argument is out of range, `backtrace-frame` returns `nil`.

0.33 Memory Allocation Changes

The list that `garbage-collect` returns now has one additional element. This is a cons cell containing two numbers. It gives information about the number of used and free floating point numbers, much as the first element gives such information about the number of used and free cons cells.

The new function `memory-limit` returns an indication of the last address allocated by Emacs. More precisely, it returns that address divided by 1024. You can use this to get a general idea of how your actions affect the memory usage.

0.34 Hook Changes

- Expanding an abbrev first runs the new hook `pre-abbrev-expand-hook`.
- The editor command loop runs the normal hook `pre-command-hook` before each command, and runs `post-command-hook` after each command.
- Auto-saving runs the new hook `auto-save-hook` before actually starting to save any files.

- The new variable `revert-buffer-insert-file-contents-function` holds a function that `revert-buffer` now uses to read in the contents of the reverted buffer—instead of calling `insert-file-contents`.
- The variable `lisp-indent-hook` has been renamed to `lisp-indent-function`.
- The variable `auto-fill-hook` has been renamed to `auto-fill-function`.
- The variable `blink-paren-hook` has been renamed to `blink-paren-function`.
- The variable `temp-buffer-show-hook` has been renamed to `temp-buffer-show-function`.
- The variable `suspend-hook` is now a normal hook. It used to be a special kind of hook; its value had to be a single function, and if the function returned a non-`nil` value, then suspension was inhibited.
- The new function `add-hook` provides a handy way to add a function to a hook variable. For example,

  ```
  (add-hook 'text-mode-hook 'my-text-hook-function)
  ```

  arranges to call `my-text-hook-function` when entering Text mode or related modes.

  `add-hook` takes an optional third argument which says to add the new hook function at the end of the list (normally, it goes at the beginning).