Macintosh®

Allegro Common LISP Dump LISP

Overview

This documentation describes the dumplisp facility in Allegro CL. This feature lets you save images (snapshots) of Allegro CL environments. Such images can be restarted very quickly. Working with images is much faster than loading fasl files.

As an example, you may generally work with the Traps and Records files loaded, and with a particular extra set of menus and Fred commands. You can arrange your system the way you like it (by loading files, etc.) and then call `dumplisp`. `dumplisp` will produce a heap image which you can later restart. The heap image will boot faster than Allegro normally boots, and it will also contain all the information from files you have loaded, etc.

To run Allegro CL with a heap image, double-click the heap image in the Finder, or select both Allegro CL and the heap image, and choose **Open** from the Finder's **File** menu. (If you use the first method, make sure there is only one copy of Allegro CL on your disk.)

Heap images created with `dumplisp` cannot be run without a copy of Allegro CL. To create stand-alone images, you need to use the Allegro Stand-alone Application generator. However, `dumplisp` is an excellent way to prototype stand-alone applications. After loading the files and setting up the menubar of your prototype stand-alone application, do a `%set-toplevel` to your toplevel function, then do a `dumplisp`. When the image file created by the dumplisp is double-clicked it will behave just like a stand-alone application.

`dumplisp` *pathname* `&key :compress :toplevel-function` [*Function*]

creates an image of the current Lisp environment and saves it to the file specified by *pathname*. If there is not enough room on the selected disk for the image, Allegro CL will exit to the Finder. In general, heap images with compression are upwards of 200K bytes and heap images without compression will be upwards of 325K bytes. These numbers will also be smaller or larger depending on how much has been loaded into the environment and whether function swapping is enabled.

pathname	a pathname for the image to be created. If there is already a file with that name, it will be deleted before the dump is performed. The default extension for images is `".image"`.
`:compress`	if true (the default), then the heap image is compressed as it is dumped. When it is rebooted, it is uncompressed in memory. Compressed images are smaller, but they take much longer to create and restart.

`:toplevel-function`
: the toplevel function to be set up when the heap image is restarted. Toplevel functions are described below. This argument defaults to the current toplevel function. If supplied, the toplevel function must be a compiled function object. (It cannot be a symbol.)

Before dumping a heap image, `dumplisp` executes all the functions in the list `*save-exit-functions*`. This list initially contains a single function which closes all windows, takes down and stores the current menubar, and disposes of other pointers to the Macintosh heap. Allegro CL then performs a garbage collection and dumps.

When a heap image is restarted, all the functions in the list `*restore-lisp-functions*` are run. This list initially contains a function which restores the menubar, restores Macintosh pointers used by the system, resets the logical pathnames `"ccl;"` and `"home;"`, and reinitializes some system configuration variables.

You can add functions to the lists `*save-exit-functions*` and `*restore-lisp-functions*`. You may wish to do this if there is state that you want to save and restore in a particular way. Items added to `*save-exit-functions*` should be added *before* the predefined function. Items added to `*restore-lisp-functions*` should always be added *after* the predefined function.

`*save-exit-functions*` [*Variable*]

a list of functions to be called when an image is dumped. These functions should perform any preparation necessary for the dump. The default list contains a single function which closes windows and performs other actions to remove pointers to the Macintosh heap. If you add functions to this list, this function should still be called last.

`*restore-lisp-functions*` [*Variable*]

a list of functions to be called when an image is restarted. These functions should perform initializations necessary to restore the system. The default list contains a single function. If you add functions, they should be placed *after* the predefined function on the list.

Macintosh Pointers

An important restriction on dumplisp is that no data on the Macintosh heap is preserved across dumps. When you perform a dump, any pointers or handles to the Macintosh heap will become invalid. For this reason, all Macintosh handles and pointers should be disposed before the dump occurs. (This is why dumplisp closes all windows before dumping.)

If your program maintains pointers to the Macintosh heap, you should deallocate these with a function included on the list `*save-exit-functions*`. You can then reinitialize the pointers and handles with a function you place on the list `*restore-lisp-functions*`.

Left over Macintosh pointers in a heap image can cause system crashes and other erratic behavior.

The Top Level Function

When Allegro CL is running, there is always a Lisp function running. For example, during normal programming, the function `toplevel-loop` runs. This function takes input from the Listener (and other buffers), evaluates it, and prints out the result.

Whenever the toplevel function returns, the Lisp kernel arranges for it to be called again. In this way Allegro CL can run indefinitely. To quit Lisp, you simply set the toplevel function to `nil`.

There are two ways to set the toplevel function. One is by making an explicit call to the function `%set-toplevel`. The other is by specifying a toplevel function when you call `dumplisp`.

Setting the Toplevel Function

Allegro CL actually has two notions of the toplevel function. One is the function which is currently running. The other is the function (stored in a variable) which will be called when the currently running toplevel function returns.

In practice, these two functions are generally the same.

When you call `%set-toplevel`, you set the pending toplevel function. This does not, however, affect the toplevel function which is currently running. The new pending toplevel function will not be called until the currently executing toplevel function returns or there is a throw to toplevel (by calling the function `toplevel`).

`%set-toplevel` `&optional` *new-top-function* [*Function*]

if *new-toplevel-function* is supplied, sets this to be the pending toplevel function and returns the previous pending toplevel function. If *new-toplevel-function* is not supplied, `%set-toplevel` simply returns the pending toplevel function.

new-top-function — a compiled function object or `nil`. This argument cannot be a symbol.

If *new-top-function* is `nil`, when the current toplevel function returns, Allegro CL will exit to the Finder.

toplevel [*Function*]

throws past all pending catches to the Lisp kernel, which then restarts the pending toplevel function. If the pending toplevel function is `nil`, the Lisp will be exited.

toplevel-loop [*Function*]

the Allegro CL function which implements the read loop. During program prototyping, it may be useful to switch between your own toplevel function and `toplevel-loop`.

Catching System Throws

You may want your toplevel function to catch errors and aborts. If you don't, errors will cause a Listener to appear, containing the error message. Catching aborts is especially important. If your toplevel loop doesn't catch them, typing command period will cause the error "Can't throw to tag `:abort`".

The following example sets up a miniature read-eval-print loop to replace the standard toplevel loop. It also shows what happens if you don't catch abort.

```
? (defun new-top (&aux form)
    (setq form (read))
    (if (eq form 'done)
          (%set-toplevel #'toplevel-loop)
        (print (eval form))))
new-top
? (%set-toplevel #'new-top)
#<Compiled Function toplevel-loop>
? (toplevel)
(+ 10 10)
20                    ;typed command-period here, to abort
> Error: Can't throw to tag :abort .
> While executing: "Unknown"
(* 20 20)
400 done
?
```

Using `*idle*`

The variable `*idle*` lets your toplevel function (or other parts of your program) Allegro CL when it is idling. If both `*idle*` and `*use-wait-next-event*` are true, Allegro CL will use the trap `_WaitNextEvent` instead of `_GetNextEvent`. This gives the maximum amount of time to background processes. During normal operation, `*idle*` is bound to `t` by the toplevel loop when it is awaiting input. Be aware that using `_WaitNextEvent` can degrade response time with older versions of the Macintosh system software.

Using `eval-enqueue`

An event (such as a menu-selection or dialog-box click) which begins a long process should not simply execute the process. If it does, the process will run with interrupts disabled, and future events will be ignored until the process returns. This is fine for quick actions, but can be a problem for time-consuming actions.

The solution to this problem is for event actions to queue up forms. The forms are received and processed in order by the toplevel loop, which keeps interrupts enabled.

There are many ways you could queue up forms. The simplest would be to just tack them onto a list, and have the toplevel loop always pop things from the list.

In many cases, you may wish to have a single mechanism which works both with your toplevel function, and with the standard toplevel function, `toplevel-loop`. To do this, you queue up forms with `eval-enqueue`. When `toplevel-loop` is running, it will automatically get these forms and evaluate them. Your toplevel loop can do the same thing by calling the `*terminal-io*` function `get-next-form`.

`get-next-form` [*Terminal IO Function*]

returns the next form from the list of all forms that have been queued up. During programming sessions, queued up forms include text entered in the Listener and evaluated from buffers, as well as forms passed to `eval-enqueue`.

Because `get-next-form` is a `*terminal-io*` object-function, you have to ask `*terminal-io*` to call it. For example:

```
(ask *terminal-io* (get-next-form))
```

If there are no pending forms, `get-next-form` will wait for a form to appear before returning. While it is awaiting input, `get-next-form` will bind `*idle*` to `t`, and it will print a Listener prompt, if necessary. If this behavior is a problem, you should check for the presence of pending forms by examining the variable `selection-queue`, before calling `get-next-form`.

`selection-queue` [*Terminal IO Variable*]

holds a list containing the queued up forms, and streams from which forms should be read. If this list is `nil`, there are no queued up forms. You generally shouldn't pop things directly from this list, but should access elements by calling `get-next-form`.

A program can test for the presence of pending forms with the call

```
(when (ask *terminal-io* selection-queue)
   . . .
   )
```

The following example shows how to replace the Allegro CL toplevel loop with a very simple read-eval-print loop:

```
? (defun mini-top (&aux form)
    (loop
       (catch-error
        (catch-abort
         (catch-cancel
          (print
           (eval
                (ask *terminal-io* (get-next-form)))))))))
mini-top
? (%set-toplevel #'mini-top)
#<Compiled Function toplevel-loop>
? (toplevel)
? (+ 5 5)

10
? *
> Error: Unbound variable: * .  ;this is only bound by
                                ;the built-in toplevel loop
> While executing: "Unknown"
? (%set-toplevel #'toplevel-loop)

#<Compiled Function mini-top>
? (toplevel)
? *
nil
```