The Stump Window Manager

Shawn Betts, David Bjergaard

1 Introduction

StumpWM is a manual, tiling X11 window manager written entirely in Common Lisp. Unlike traditional window managers, StumpWM places windows in order to maximize the amount of the screen used. The window layouts managed by StumpWM are defined by the user in much the same way that windows are managed by GNU screen, or emacs.

Before StumpWM, there was ratpoison, another tiling window manager written entirely in C. StumpWM grew out of the authors' frustration with writing ratpoison in C. Very quickly we realized we were building into ratpoison lispy-emacs style paradigms. StumpWM's goals are similar to ratpoison's but with an emphasis on customizability, completeness, and cushiness.

1.1 Starting StumpWM

There are a number of ways to start StumpWM but the most straight forward method is as follows. This assumes you have a copy of the StumpWM source code and are using the '`SBCL`' Common Lisp environment.

1. Install the prerequisites and build StumpWM as described in `README`. This should give you a `stumpwm` executable.
2. In your `~/.xinitrc` file include the line `/path/to/stumpwm`. Remember to replace '`/path/to/`' with the actual path.
3. Finally, start X windows with `startx`. Cross your fingers. You should see a '`Welcome To the Stump Window Manager`' message pop up in the upper, right corner. At this point, you have successfully started StumpWM.

1.2 Basic Usage

Once you have StumpWM up and running, the first thing you might want to do is start `emacs`. Type `C-t e`, or in other words `Control + t` followed by `e`. Now perhaps you want an `xterm`. Type `C-t c`. Now you have some programs running.

To see a list of windows StumpWM is managing, type `C-t w`. The highlighted window is the one you're looking at right now. It's the focused window.

All of StumpWM's keys are bound to named commands, which can be executed not only by keys but also from the input bar. Type `C-t ;` to open a command prompt. Now type `time` and press return. Note, `time` can also be called by typing `C-t a`. Throughout this manual you'll find definitions for commands, functions, and variables. Any command you see in this manual can be executed from the input bar or bound to a key.

At this point you probably want to switch back from your new `xterm` to `emacs`. Type `C-t C-t`. This runs the `other` command. Type it again and you're back to xterm.

Perhaps you'd like to see `emacs` and `xterm` side-by-side. Type `C-t s`. You have now split the screen into 2 `frames`. For more information see Chapter 6 [Frames], page 27. To switch to the empty frame type `C-t TAB`. Now let's pull the xterm window into this empty frame. Type `C-t w` for a window listing. Find the `xterm` window. See the number beside it? Type `C-t` followed by `xterm`'s window number.

Another common activity is browsing the internet. Type `C-t !`. The input bar pops up again. You can now run a shell command. Let's start a web browser: type `firefox` into the input bar and press return.

Unfortunately, `firefox` probably isn't wide enough because it's in one of the frames. Type `C-t Q` to remove all frames but the current one and resize it to fit the screen.

For a full list of key bindings, see Section 2.1 [List of Default Keybindings], page 7.

1.3 Basic Concepts

An introduction to some of the basic concepts used by StumpWM.

1.3.1 Screens and Heads

A screen is an Xlib concept representing a section of video memory onto which physical monitors, called "heads", are mapped. A screen can be thought of as an abstract rectangle containing all the heads arranged in a particular layout.

With most modern systems, you'll only have a single screen no matter how many heads are connected to your computer. Each head will have its own frame, and you can move between heads using the normal frame movement commands.

The layout of the heads within the screen can be specified in one of two ways: either at startup using your system's Xorg configuration files, or on the fly using tools like XRandR. If the computer is booted with multiple monitors attached, but without specifying a layout for them, they will all show identical output.

StumpWM will attempt to detect the layout of the heads once at startup, or any time a RandR command is issued.

In rarer setups you may have multiple screens, with one head per screen. That means that you'll move between heads using screen movement commands ('snext', 'sprev', and 'sother') rather than frame movement commands.

1.3.2 Group Basics

A group is usually referred to as a "desktop" or "workspace" in other window managers. StumpWM starts with a single group, called "Default". Each group has its own configuration of frames and windows that is separate from and independent of other groups. You can't have different groups display in different monitors: when you switch groups, all monitors switch to that group.

Each group contains an ordered list of frames.

1.3.3 Floating Group Basics

Within a floating group, windows behave more like they do in traditional window managers: rather than being arranged into frames, they each have their own box, which can be freely resized and repositioned, and allowed to overlap. Each window has a thicker border at the top. Left click in this border and drag to move the window, or right click and drag to resize it.

Most of the window-switching commands listed below do not function in a floating group. You're restricted to 'other', the 'select-window-*' commands, and 'windowlist'.

1.3.4 Frame Basics

Frames are the boxes within which windows are displayed. StumpWM starts with a single frame per head, meaning that each monitor shows a single window, full screen. If you want to see windows side-by-side, you can "split" this frame in two, either vertically or horizontally. These frames can be further split, creating nested boxes.

Technically speaking, frames live within a "frame tree". When you split a frame, the command actually creates *two* new frames side-by-side within the original parent frame. This makes no practical difference, unless you use the 'sibling' command, which will move to the other child frame within the parent frame.

Within this frame tree model, all frames either contain other frames, or windows. The command 'fclear' will hide all a frame's windows and show the background.

1.3.5 Window Basics

Windows are created by programs to display their output. They take the shape of the frame in which they are created. The windows within a frame are ordered by how recently that window was focused. Only the top window in the stack is visible.

1.3.6 System Trays and the Mode Line

Many users choose to sacrifice a little screen real-estate to display some generally useful information: the current time and date, wireless network connections, the names of open windows, etc. StumpWM allows you to display this information in a bar across either the top or the bottom of the screen. There are two ways to do this: using external programs called system trays, or using StumpWM's own mode line.

System trays are a special kind of X window. They advertise to running programs that they are available for embedding icons or notifications from those programs. They often also display clickable icons for each open window. Common tray programs include the GNOME panel or KDE's kicker, or simpler programs such as stalonetray. Simply starting one of these programs is usually enough for StumpWM to detect it, place it correctly, and allow it to function normally.

The mode line, a concept borrowed from Emacs, is a built-in part of StumpWM. It is essentially a string of text that can include a variety of information about your current session, including the names of current groups and windows. Several modules provide for different types of information. See Chapter 7 [Mode-line], page 31, (and the modules directory) for more.

1.4 Manipulating Frames and Windows

Frames and windows are concepts borrowed from Emacs and the GNU Screen program, and should be familiar to users of those programs. Others may find the terms a little confusing. In other window managers, a "window" usually refers to a bounded box on the screen, showing output from a single program. StumpWM splits this into two concepts: the "frame" is the bounded box, the "window" is the visible output of a program.

One frame can contain many windows. As new windows are created, they appear at the top of the window-stack of the current frame. This is also a little different from other tiling window managers, many of which automatically create new frames for new windows.

Both frames and windows are ordered by when they were last focused. In the following commands and documentation, the terms "next" and "previous" refer to this order. "Other" refers to the most-recently focused object. Calling "other" commands multiple times will bounce back and forth between the two most recent objects.

By default, StumpWM starts with a single group, called "Default", which contains one full-screen frame per head. You can split individual frames horizontally or vertically using the 'hsplit' and 'vsplit' commands, bound to "C-t S" and "C-t s" by default. When a frame is split, the next-most-recently-focused window is pulled into the new frame. See Chapter 6 [Frames], page 27, and Chapter 5 [Windows], page 21, for a complete listing of commands.

1.4.1 Moving Between Frames

Once you have multiple frames, you can move between them in various ways:

- `fnext` (`C-t o` or `C-t TAB`) jumps to the next frame in the current group's frame list.
- `fother` (`C-t M-TAB`) jumps to the last frame that had focus.
- `fselect` (`C-t f`) displays numbers on each visible frame: hit a number key to move to that frame.
- `move-focus` (`C-t <arrow key>`) focus the frame in the direction of the arrow key pressed.
- `sibling` (unbound by default) focus the frame from which the current frame was split.

1.4.2 Manipulating Windows

Some commands change which window is currently focused, some move windows between frames, and some may do both at once.

There are two general ways to move focus between windows: either between windows belonging to the current frame, or between all windows within the current group. Within a single frame:

- `next-in-frame` (`C-t C-M-n`) focus the next window in the current frame's list of windows.
- `prev-in-frame` (`C-t C-M-p`) focus the previous window in the current frame's list of windows.
- `other-in-frame` (`C-t M-t`) focus the most recently focused window in the current frame's list of windows.
- `frame-windowlist` (unbound by default) display a menu of windows in the currently-focused frame, and allow the user to choose one. Alternately, the command `frame-windows` will simply display the list of window names, with no menu choice available.

Within the current group, the following commands will go straight to the specified window. They will never move a window from its original frame, and so may result in focus switching frames.

- `next` (`C-t M-n`) focus the next window in the current group.
- `prev` (`C-t M-p`) focus the previous window in the current group.
- `other` or `other-window` (unbound by default) focus the most recently focused window in the current group.
- `next-urgent` (`C-t C-u`) focus the next window that has marked itself "urgent".

- `select` or `select-window` (`C-t '`) prompt for the title of a window and focus it. Works with partial completion of the title.
- `select-window-by-name` (unbound by default) prompt for the title of a window and focus it. Requires the window title to be entered exactly.
- `select-window-by-number` (`C-t <number>`) choose a window by number.
- `windowlist` ("C-t `""`) display a menu of windows in the currently-focused group, and allow the user to choose one.

The following commands always keep the current frame focused. If the selected window is not in the current frame, it will be pulled there from wherever it is (hence the "pull" naming scheme).

- `pull` or `pull-window-by-number` (`C-t C-<number>`) pull the numbered window into the current frame.
- `pull-hidden-next` (`C-t n` or `C-t SPC`) pull the next currently undisplayed window in the window list into the current frame.
- `pull-hidden-previous` (`C-t p`) pull the previous currently undisplayed window in the window list into the current frame.
- `pull-hidden-other` (`C-t C-t`) pull the most recently focused, currently undisplayed window into the current frame.

The following commands move the current window from one frame to another, bringing focus with them.

- `move-window` (`C-t M-<arrow>`) move the currently focused window in the direction indicated by the arrow key.
- `exchange-direction` (unbound by default) prompt for a direction, then swap the currently focused window with the top window of the frame in that direction.

1.5 Interacting with the Lisp process

Since StumpWM is a Lisp program, there is a way for you to evaluate Lisp code directly, on the same Lisp process that StumpWM is running on. Type `C-t :` and an input box will appear. Then type some Lisp expression.

When you call `eval` this way, you will be in the STUMPWM-USER package, which imports all the exported symbols from the main STUMPWM package.

`*mode-line-border-width*`
: Reads the value of **mode-line-border-width**.

`(setf *mode-line-border-width* 3)`
: Sets the variable **mode-line-border-width** to 3.

`(set-prefix-key (kbd "C-M-H-s-z"))`
: Calls the `set-prefix-key` function (and sets a new keyboard prefix)

1.6 Contact the StumpWM developers

The StumpWM home page is `http://stumpwm.nongnu.org/`.

You can reach Shawn Betts at `sabetts at vcn.bc.ca`.

The StumpWM mailing list is `stumpwm-devel@nongnu.org` which you can subscribe to at `http://mail.nongnu.org/mailman/listinfo/stumpwm-devel`. Posting is restricted to subscribers to keep spam out of the archives.

The StumpWM IRC channel can be found on Freenode at `#stumpwm` (`irc://irc.freenode.net/#stumpwm`).

2 Key Bindings

StumpWM is controlled entirely by keystrokes and Lisp commands. It mimics GNU Screen's keyboard handling. StumpWM's default prefix key is `C-t`.

2.1 List of Default Keybindings

The following is a list of keybindings.

`C-t` *d* — Select the window with the corresponding digit *d*

`C-t C-`*d* — Pull the window with the corresponding digit *d* into the current frame

`C-t n`
`C-t C-n`
`C-t Space` — Go to the next window in the window list

`C-t p`
`C-t C-p` — Go to the previous window in the window list

`C-t '` — Go to a window by name

`C-t "` — Select a window from a list and focus the window.

`C-t C-g` — Abort the current command. This is useful if you accidentally hit `C-t`

`C-t w` — List all the windows

`C-t i` — Display information about the current window.

`C-t f` — Select a frame by number

`C-t s` — Split current frame vertically

`C-t S` — Split current frame horizontally

`C-t k`
`C-t C-k` — Sends a kill message to the current frame and the running program.

`C-t K` — Kills the current frame and running program; like a `kill -9`.

`C-t c`
`C-t C-c` — Run an X terminal; by default `xterm`

`C-t e`
`C-t C-e` — Run Emacs or raise it if it is already running.

`C-t t` — Sends a `C-t` to the frame; this is useful for applications like Firefox which make heavy use of `C-t` (in Firefox's case, for opening a new tab). This is similar to how GNU screen uses `C-a a`.

`C-t w`
`C-t C-w` — Prints out a list of the windows, their number, and their name.

`C-t b`
`C-t C-b` — Banish the mouse point to the lower right corner of the screen.

`C-t a`
`C-t C-a` — Display the current time and date, much like the Unix command `date`.

`C-t C-t` Switch to the last window to have focus in the current frame.

`C-t !` Prompt for a shell command to run via `/bin/sh`. All output is discarded.

`C-t R` If the screen is split into multiple frames, one split will be undone. If there is only one split, the effect will be the same as `C-t Q`.

`C-t o`
`C-t TAB` If the screen is split into multiple frames, focus shifts to the `next` frame, where it cycles to the right and then down; analogous to `C-x o` in Emacs.

`C-t F` Display "Current Frame" in the frame which has focus.

`C-t ;` Opens the input box. StumpWM commands can be run from here, and the input history moved through.

`C-t :` Opens the input box, but all things typed in here will be sent to the Common Lisp interpreter where they will be ran as Lisp programs; thus, input should be valid Common Lisp.

`C-t C-h`
`C-t ?` The help.

`C-t -` Hide all frames and show the root window.

`C-t Q` Removes all splits and maximizes the frame with focus.

`C-t Up`
`C-t Down`
`C-t Left`
`C-t Right` Shift focus to an adjacent frame in the specified direction. `C-t Up` will shift focus up, if possible, `C-t Down` will shift downwards, etc.

`C-t v` Prints out the version of the running StumpWM.

`C-t #` Toggle the mark on the current window

`C-t m`
`C-t C-m` Display the last message. Hitting this keybinding again displays the message before that, and so on.

`C-t l`
`C-t C-l` redisplay the current window and force it to take up the entire frame.

`C-t G` Display all groups and windows in each group. For more information see Chapter 8 [Groups], page 33.

`C-t F`*n* Jump to the corresponding group *n*. `C-t F1` jumps to group 1 and so on.

`C-t g g` Show the list of groups.

`C-t g c` Create a new group.

`C-t g n`
`C-t g C-n`
`C-t g SPC`
`C-t g C-SPC`
Go to the next group in the list.

`C-t g N` Go to the next group in the list and bring the current window along.

`C-t g p`
`C-t g C-p` Go to the previous group in the list.

`C-t g P` Go to the previous group in the list and bring the current window along.

`C-t g '` Select a group by name or by number.

`C-t g "` Select a group from a list and switch to it.

`C-t g m` Move the current window to the specified group.

`C-t g k` Kill the current group. All windows are merged into the next group.

`C-t g A`
`C-t g r` Change the current group's name.

`C-t g` *d* Go to the group with digit *d*. `C-t g 1` jumps to group 1 and so on.

`C-t +` Make frames the same height or width in the current frame's subtree.

`C-t h k` Describe the specified key binding.

`C-t h f` Describe the specified function.

`C-t h v` Describe the specified variable.

`C-t h w` List all key sequences that are bound to the specified command

`C-t h c` Describe the specified command.

2.2 Binding Keys

`define-key` *map key command* [Function]

Add a keybinding mapping for the key, *key*, to the command, *command*, in the specified keymap. If *command* is nil, remove an exising binding. For example,

```
(stumpwm:define-key stumpwm:*root-map* (stumpwm:kbd "C-z") "echo Zzzzz...")
```

Now when you type C-t C-z, you'll see the text "Zzzzz..." pop up.

`undefine-key` *map key* [Function]

Clear the key binding in the specified keybinding.

`kbd` *keys* [Function]

This compiles a key string into a key structure used by 'define-key', 'undefine-key', 'set-prefix-key' and others.

`set-prefix-key` *key* [Command]

Change the stumpwm prefix key to KEY.

```
(stumpwm:set-prefix-key (stumpwm:kbd "C-M-H-s-z"))
```

This will change the prefix key to `Control + Meta + Hyper + Super +` the `z` key. By most standards, a terrible prefix key but it makes a great example.

`make-sparse-keymap` [Function]

Create an empty keymap. If you want to create a new list of bindings in the key binding tree, this is where you start. To hang frame related bindings off `C-t C-f` one might use the following code:

```
(defvar *my-frame-bindings*
  (let ((m (stumpwm:make-sparse-keymap)))
    (stumpwm:define-key m (stumpwm:kbd "f") "curframe")
    (stumpwm:define-key m (stumpwm:kbd "M-b") "move-focus left")
    m ; NOTE: this is important
  ))

(stumpwm:define-key stumpwm:*root-map* (stumpwm:kbd "C-f") '*my-frame-bindings*)
```

`*root-map*` [Variable]

This is the keymap by default bound to `C-t`. It is known as the *prefix map*.

`*top-map*` [Variable]

The top level key map. This is where you'll find the binding for the *prefix map*.

`*groups-map*` [Variable]

The keymap that group related key bindings sit on. It is bound to `C-t g` by default.

`*exchange-window-map*` [Variable]

The keymap that exchange-window key bindings sit on. It is bound to `C-t x` by default.

`bind` *key command* [Command]

Hang a key binding off the escape key.

2.3 Modifiers

Many users have had some difficulty with setting up modifiers for StumpWM keybindings. This is caused by a combination of how StumpWM handles modifiers and the default modifiers list for many users' X servers.

- My "Super" key doesn't work!

  This is most likely caused by having the Hyper and Super keys listed as the same modifier in the modifier list.

```
$ xmodmap
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x32),  Shift_R (0x3e)
lock        Caps_Lock (0x42)
control     Control_L (0x25),  Control_R (0x6d)
mod1        Alt_L (0x40),  Alt_R (0x71),  Meta_L (0x9c)
mod2        Num_Lock (0x4d)
mod3
mod4        Super_L (0x7f),  Hyper_L (0x80)
mod5        Mode_switch (0x5d),  ISO_Level3_Shift (0x7c)
```

The problem is in the line beginning with "mod4". The way to set up the modifier list correctly is to have just the Super key as the mod4 modifier. The following `xmodmap` commands will do just that.

```
# clear out the mod4 modifier
$ xmodmap -e 'clear mod4'
$ xmodmap
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x32),  Shift_R (0x3e)
lock        Caps_Lock (0x42)
control     Control_L (0x25),  Control_R (0x6d)
mod1        Alt_L (0x40),  Alt_R (0x71),  Meta_L (0x9c)
mod2        Num_Lock (0x4d)
mod3
mod4
mod5        Mode_switch (0x5d),  ISO_Level3_Shift (0x7c)

# add Super as a mod4 modifier
$ xmodmap -e 'add mod4 = Super_L'
$ xmodmap
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x32),  Shift_R (0x3e)
lock        Caps_Lock (0x42)
control     Control_L (0x25),  Control_R (0x6d)
mod1        Alt_L (0x40),  Alt_R (0x71),  Meta_L (0x9c)
mod2        Num_Lock (0x4d)
mod3
mod4        Super_L (0x73),  Super_L (0x7f)
mod5        Mode_switch (0x5d),  ISO_Level3_Shift (0x7c)
```

You can automate this by storing the commands in a file and calling xmodmap when you start your X session.

```
$ cat ~/.Xmodmap
clear mod4
add mod4 = Super_L
```

If you use `startx`, modify your `~/.xsession` or `~/.xinitrc` file.

```
$ cat ~/.xsession
#!/bin/sh

xmodmap ~/.Xmodmap
exec /usr/bin/stumpwm
```

If you use a settings daemon from one of the major desktop environments (Gnome,KDE, or Unity) you may be able to set keyboard modifiers from their respective configuration GUIs. If not, `xmodmap` should always work if invoked at the right place.

- Handling Meta and Alt: when do I use `M-` and `A-`?

If you have no Meta keys defined (see the output of the `xmodmap` command), then StumpWM will treat the `M-` prefix in keybindings to mean Alt. However, if there are Meta keys defined, then the `M-` prefix refers to them, and the `A-` prefix refers to Alt.

Most users will simply use `M-` to refer to their Alt keys. However, users that define separate Meta and Alt keys will use `M-` to refer to the former, and `A-` to refer to the latter.

- How can I set up a Hyper key and use it with StumpWM?

  To set up a Hyper key, you need to do two things: bind a physical key to be a Hyper key, and add that key to the modifiers list.

  The following example shows how to bind the control key at the bottom-left of most keyboards to be Hyper. This is useful if you've made Caps Lock into a control key, and have no use for the bottom-left key.

  ```
  $ xmodmap -e 'keycode 37 = Hyper_L'
  $ xmodmap -e 'clear mod5'
  $ xmodmap -e 'add mod5 = Hyper_L'
  ```

  To use a different key for Hyper, replace the keycode "37" above. Use the `xev` program to see the keycode that any physical key has. Refer to the section above on setting up the Super key to see how to automate setting the Hyper key when you start X.

  Now you can use `H-` as a prefix in StumpWM bindings.

  ```
  (define-key *top-map* (kbd "H-RET") "fullscreen")
  (define-key *top-map* (kbd "H-Left") "gprev")
  (define-key *top-map* (kbd "H-Right") "gnext")
  (define-key *top-map* (kbd "H-TAB") "other")
  ```

  Since essentially no programs have Hyper bindings, you can safely bind commands to the **top-map**.

3 Commands

If you've used Emacs before you'll find the distinction between commands and functions familiar. Commands are simply functions that can be bound to keys and executed interactively from StumpWM's input bar. Whereas, in Emacs, the special `"(interactive)"` declaration is used to turn a function into a command, in StumpWM commands are made with a separate `defcommand` macro.

Once a command is defined, you can call it by invoking the `colon` command (`C-t ;`), and typing the name of the command. This may be sufficient for commands that aren't used very often. To see all the currently-defined commands, invoke the command called `commands`: ie press `C-t ;`, type "commands", and hit return.

Commonly-used commands can also be bound to a keystroke, which is much more convenient. To do this, use the `define-key` function (see Chapter 2 [Key Bindings], page 7), giving the name of the command as a string. For example:

```
(define-key *root-map* (kbd "d") "exchange-direction")
```

You cannot give the command name as a symbol, nor can you bind a key to a regular function defined with `defun`.

If the command takes arguments (see Section 3.1 [Writing Commands], page 13), you can fix those arguments when defining the key-binding, by including the arguments in the same string as the command name, separated by a space. For instance, the `exchange-direction` command, which is unbound by default, requires a direction in which to exchange windows. If you call `exchange-direction` directly, it will prompt you for the direction. If you know that you often exchange in left/right directions, and want those actions bound to keys, you can use the following in your customization file:

```
(define-key *root-map* (kbd "[") "exchange-direction left")
(define-key *root-map* (kbd "]") "exchange-direction right")
```

Multiple arguments can be included by adding them to the command string, separated by spaces. Not all argument types can be represented as strings, but StumpWM will do its best to convert types.

StumpWM does not implement the Emacs concept of prefix arguments.

3.1 Writing Commands

StumpWM commands are written much like any Lisp function. The main difference is in the way command arguments are specified. The `defcommand` macro takes a list of arguments as its first form (similar to the `defun` macro), and a corresponding list of types as its second form. All arguments must belong to a "type". Each type specification has two parts: a keyword specifying the argument type, and a string prompt that will be displayed when asking the user to enter the argument value. A typical `defcommand` might look like this:

```
(defcommand now-we-are-six (name age)
    ((:string "Enter your name: ")
     (:number "Enter your age: "))
  (message "~a, in six years you will be ~a" name (+ 6 age)))
```

If `now-we-are-six` is called interactively via the `colon` command, the user will be prompted for a string and a number, which will then be bound to "name" and "age", respectively, in the body of the command.

When invoking the command via a key-binding, it is possible to provide some or all of the arguments directly:

```
(define-key *root-map* (kbd "L") "now-we-are-six John")
```

In this case, hitting `C-t L` will only prompt for an age (the first string argument is already bound to "John"). Argument values provided this way always bind to the earliest arguments defined: ie, it is not possible to specify an age, but prompt the user for a name.

If the type declaration does not include a prompt (ie, it looks like "(:type nil)", or "(:type)" or just ":type"), the argument is considered optional. It can be provided via a key-binding invocation, as above, but if it isn't, the user will not be prompted, and the argument will be bound to nil.

Lastly, it is possible to limit the scope under which the command will be usable: a command can be defined to work only in tile groups, or only in floating groups (the only two types of groups that currently exist). This is done by replacing the name of the command with a two-element list: the name of the command as a symbol, and either the symbol tile-group or floating-group. For instance, the `next` command, which only functions in tile groups, is defined this way:

```
(defcommand (next tile-group) ...)
```

3.2 StumpWM Types

All command arguments must be of a defined "StumpWM type". The following types are pre-defined:

:y-or-n A yes or no question returning T or NIL.

:variable A lisp variable

:function A lisp function

:command A stumpwm command as a string.

:key-seq A key sequence starting from *TOP-MAP*

:window-number
An existing window number

:number An integer number

:string A string

:key A single key chord

:window-name
An existing window's name

:direction A direction symbol. One of :UP :DOWN :LEFT :RIGHT

:gravity A gravity symbol. One of :center :top :right :bottom :left :top-right :top-left :bottom-right :bottom-left

:group An existing group

:frame A frame

:shell A shell command

:rest The rest of the input yes to be parsed.

:module An existing stumpwm module

Additional types can be defined using the macro `define-stumpwm-type`. Emacs users who are accustomed to writing more complicated interactive declarations using "(interactive (list ...))" forms will find that similar logic can be put into StumpWM type definitions. The macro is called like this:

(define-stumpwm-type :type-name (input prompt) body)

The keyword :type-name will then be available for use in `defcommand` macros. When commands are called, the bodies of these type definitions are called in turn to produce actual argument values.

Type definitions produce their value in one of several ways: by reading it from the argument line bound to a keystroke, by prompting the user to enter a value, or by generating it programmatically.

Within the body of the type definition, the argument "input" is bound to the argument line provided in the command string, and "prompt" to the string prompt provided in the `defcommand` form. The usual convention is to first check if an argument has been provided in "input" and, if it hasn't, to prompt for it using "prompt".

StumpWM provides several convenience functions for handling the value of "input":

- `argument-pop` (input) pop the next space-delimited argument from the argument line.
- `argument-pop-rest` (input) return the remainder of the argument line as a single string, leaving input empty
- `argument-pop-or-read` (input prompt &optional completions) either pop an argument from the argument line, or if it is empty use "prompt" to prompt the user for a value
- `argument-pop-rest-or-read` (input prompt &optional completions) either return the remainder of the argument line as a string, leaving input empty, or use "prompt" to prompt the user for a value

As an example, here's a new type called :smart-direction. The existing :direction type simply asks for one of the four directions "left", "right", "up" or "down", without checking to see if there's a frame in that direction. Our new type, :smart-direction, will look around the current frame, and only allow the user to choose a direction in which another frame lies. If only one direction is possible it will return that automatically without troubling the user. It signals an error for invalid directions; it could alternately return a "nil" value in those cases, and let the command handle that.

```
(define-stumpwm-type :smart-direction (input prompt)
  (let ((valid-dirs
         (loop  ; gather all the directions in which there's a neighbouring frame
            with values = '(("up" :up)
                            ("down" :down)
                            ("left" :left)
                            ("right" :right))
            with frame-set =
              (group-frames (window-group (current-window)))
            for dir in values
```

```
            for neighbour = (neighbour
                             (second dir)
                             (window-frame (current-window)) frame-set)
            if (and neighbour (frame-window neighbour))
            collect dir))
        (arg (argument-pop input)))  ; store a possible argument
    (cond ((null valid-dirs)  ; no directions, bail out
           (throw 'error "No valid directions"))
          (arg  ; an arg was bound, but is it valid?
           (or (second (assoc arg valid-dirs :test #'string=))
               (throw 'error "Not a valid direction")))
          ((= 1 (length valid-dirs))  ; only one valid direction
           (second (car valid-dirs)))
          (t  ; multiple possibilities, prompt for direction
           (second (assoc (completing-read input prompt valid-dirs
                                           :require-match t)
                          valid-dirs :test #'string=))))))

(defcommand smarty (dir) ((:smart-direction "Pick a direction: "))
  ;; `dir' is a keyword here
  (message "You're going ~a" (string-downcase dir)))

(define-key *root-map* (kbd "R") "smarty right")
```

4 Message and Input Bar

`echo` *string* [Command]
Display *string* in the message bar.

`colon` **&optional** *initial-input* [Command]
Read a command from the user. *initial-text* is optional. When supplied, the text will appear in the prompt.

4.1 Customizing The Bar

The bar's appearance and behavior can be modified with the following functions and variables. See Chapter 13 [Colors], page 49, for an explanation of how to set these color variables.

`set-fg-color` *color* [Function]
Set the foreground color for the message bar and input bar. *color* can be any color recognized by X.

`set-bg-color` *color* [Function]
Set the background color for the message bar and input bar. *color* can be any color recognized by X.

`set-border-color` *color* [Function]
Set the border color for the message bar and input bar. *color* can be any color recognized by X.

`set-msg-border-width` *width* [Function]
Set the border width for the message bar, input bar and frame indicator.

`set-font` *font* [Function]
Set the font(s) for the message bar and input bar.

`*message-window-padding*` [Variable]
The number of pixels that pad the text in the message window.

`*message-window-gravity*` [Variable]
This variable controls where the message window appears. The follow are valid values.

:top-left

:top-right

:bottom-left
:bottom-right
:center

:top

:left

:right

:bottom

`*timeout-wait*` [Variable]
Specifies, in seconds, how long a message will appear for. This must be an integer.

`*input-window-gravity*` [Variable]
This variable controls where the input window appears. The follow are valid values.

:top-left

:top-right

:bottom-left
:bottom-right
:center

:top

:left

:right

:bottom

4.2 Using The Input Bar

The following is a list of keybindings for the Input Bar. Users of Emacs will recognize them.

`DEL` Delete the character before point (`delete-backward-char`).

`M-DEL` Kill back to the beginning of the previous word (`backward-kill-word`).

`C-d`
`Delete` Delete the character after point (`delete-forward-char`).

`M-d` Kill forward to the end of the next word (`forward-kill-word`).

`C-f`
`Right` Move forward one character (`forward-char`).

`M-f` Move forward one word (`forward-word`).

`C-b`
`Left` Move backward one character (`backward-char`).

`M-b` Move backward one word (`backward-word`).

`C-a`
`Home` Move to the beginning of the current line (`move-beginning-of-line`).

`C-e`
`End` Move to the end of the current line (`move-end-of-line`).

`C-k` Kill to the end of the line (`kill-line`).

`C-u` Kill to the beginning of the line (`kill-to-beginning`), the same as `C-a C-k`.

`C-p`
`Up` Move to the next earlier entry saved in the command history (`history-back`).

`C-n`
`Down` Move to the next later entry saved in the command history (`history-forward`).

`RET` Submit the entered command (`submit`).

`C-g` Abort the current action by closing the Input Bar (`abort`).

`C-y` Paste text from clipboard into the Input Bar (`yank-selection`).

`TAB` Clockwise tab complete the current string, if possible. Press `TAB` again to cycle through completions.

`S-TAB` Counter-clockwise tab complete the current string, if possible. Press `S-TAB` again to cycle through completions.

4.3 Programming The Message Bar

`echo-string` *screen msg* [Function]
Display *string* in the message bar on *screen*. You almost always want to use `message`.

`message` *fmt* **&rest** *args* [Function]
run FMT and ARGS through 'format' and echo the result to the current screen.

`*input-history-ignore-duplicates*` [Variable]
Do not add a command to the input history if it's already the first in the list.

`copy-last-message` [Command]
Copy the last message displayed into the X selection

4.4 Programming the Input Bar

New input behavior can be added to the input bar by creating editing functions and binding them to keys in the **input-map** using `define-key`, just like other key bindings.

An input function takes 2 arguments: the input structure and the key pressed.

`read-one-line` *screen prompt* **&key** (*initial-input*) *require-match* *password* [Function]
Read a line of input through stumpwm and return it. returns nil if the user aborted.

`read-one-char` *screen* [Function]
Read a single character from the user.

`completing-read` *screen prompt completions* **&key** (*initial-input*) *require-match* [Function]
Read a line of input through stumpwm and return it with TAB completion. completions can be a list, an fbound symbol, or a function. if its an fbound symbol or a function then that function is passed the substring to complete on and is expected to return a list of matches. If require-match argument is non-nil then the input must match with an element of the completions.

`input-insert-string` *input string* [Function]
Insert *string* into the input at the current position. *input* must be of type *input-line*. Input functions are passed this structure as their first argument.

`input-insert-char` *input char* [Function]

Insert *char* into the input at the current position. *input* must be of type *input-line*. Input functions are passed this structure as their first argument.

`*input-map*` [Variable]

This is the keymap containing all input editing key bindings.

5 Windows

`next` [Command]
Go to the next window in the window list.

`pull-hidden-next` [Command]
Pull the next hidden window into the current frame.

`prev` [Command]
Go to the previous window in the window list.

`pull-hidden-previous` [Command]
Pull the next hidden window into the current frame.

`delete-window` **&optional** (*window* **(current-window)**) [Command]
Delete a window. By default delete the current window. This is a request sent to the window. The window's client may decide not to grant the request or may not be able to if it is unresponsive.

`kill-window` **&optional** (*window* **(current-window)**) [Command]
Tell X to disconnect the client that owns the specified window. Default to the current window. if `delete-window` didn't work, try this.

`echo-windows` **&optional** (*fmt* ***window-format***) (*group* [Command]
(current-group)) (windows (group-windows group)) Display a list of managed windows. The optional argument *fmt* can be used to override the default window formatting.

`other-window` **&optional** (*group* **(current-group)**) [Command]
Switch to the window last focused.

`pull-hidden-other` [Command]
Pull the last focused, hidden window into the current frame.

`renumber` *nt* **&optional** (*group* **(current-group)**) [Command]
Change the current window's number to the specified number. If another window is using the number, then the windows swap numbers. Defaults to current group.

`meta` *key* [Command]
Send a fake key to the current window. *key* is a typical StumpWM key, like `C-M-o`.

`select-window` *query* [Command]
Switch to the first window that starts with *query*.

`select-window-by-number` *num* **&optional** (*group* **(current-group)**) [Command]
Find the window with the given number and focus it in its frame.

`title` *title* [Command]
Override the current window's title.

`windowlist` **&optional** (*fmt* ***window-format***) [Command]
Allow the user to Select a window from the list of windows and focus the selected window. For information of menu bindings See Section 12.1 [Menus], page 45. The optional argument *fmt* can be specified to override the default window formatting.

`fullscreen` [Command]
Toggle the fullscreen mode of the current widnow. Use this for clients with broken (non-NETWM) fullscreen implemenations, such as any program using SDL.

`info` **&optional** (*fmt* ***window-info-format***) [Command]
Display information about the current window.

`refresh` [Command]
Refresh current window without changing its size.

`redisplay` [Command]
Refresh current window by a pair of resizes, also make it occupy entire frame.

`*window-format*` [Variable]
This variable decides how the window list is formatted. It is a string with the following formatting options:

%n Substitutes the windows number translated via *window-number-map*, if there are more windows than *window-number-map* then will use the window-number.

%s Substitute the window's status. * means current window, + means last window, and - means any other window.

%t Substitute the window's name.

%c Substitute the window's class.

%i Substitute the window's resource ID.

%m Draw a # if the window is marked.

Note, a prefix number can be used to crop the argument to a specified size. For instance, '`%20t`' crops the window's title to 20 characters.

`*window-name-source*` [Variable]
This variable controls what is used for the window's name. The default is `:title`.

`:title` Use the window's title given to it by its owner.

`:class` Use the window's resource class.

`:resource-name`
Use the window's resource name.

`*new-window-prefered-frame*` [Variable]
nil

5.1 Window Marks

Windows can be marked. A marked window has a # beside it in the window list. Some commands operate only on marked windows.

`mark` [Command]
Toggle the current window's mark.

`clear-window-marks` **&optional** (*group* **(current-group)**) (*windows* [Command]
(group-windows group)) Clear all marks in the current group.

`pull-marked` [Command]
Pull all marked windows into the current frame and clear the marks.

5.2 Customizing Window Appearance

`*maxsize-border-width*` [Variable]
The width in pixels given to the borders of windows with maxsize or ratio hints.

`*transient-border-width*` [Variable]
The width in pixels given to the borders of transient or pop-up windows.

`*normal-border-width*` [Variable]
The width in pixels given to the borders of regular windows.

`*window-border-style*` [Variable]
This controls the appearance of the border around windows. valid values are:

:thick All space within the frame not used by the window is dedicated to the border.

:thin Only the border width as controlled by *maxsize-border-width* *normal-border-width* and *transient-border-width* is used as the border. The rest is filled with the unfocus color.

:tight The same as :thin but the border surrounds the window and the wasted space within the frame is not obscured, revealing the background.

:none Like :tight but no border is ever visible.

After changing this variable you may need to call sync-all-frame-windows to see the change.

See Chapter 13 [Colors], page 49, for an explanation of how to set these color variables.

`set-win-bg-color` *color* [Function]
Set the background color of the window. The background color will only be visible for windows with size increment hints such as '`emacs`' and '`xterm`'.

`set-focus-color` *color* [Function]
Set the border color for focused windows. This is only used when there is more than one frame.

`set-unfocus-color` *color* [Function]
Set the border color for windows without focus. This is only used when there is more than one frame.

`set-float-focus-color` *color* [Function]
Set the border color for focused windows in a float group.

`set-float-unfocus-color` *color* [Function]
Set the border color for windows without focus in a float group.

`set-normal-gravity` *gravity* [Function]
Set the default gravity for normal windows. Possible values are `:center` `:top` `:left` `:right` `:bottom` `:top-left` `:top-right` `:bottom-left` and `:bottom-right`.

`set-maxsize-gravity` *gravity* [Function]
Set the default gravity for maxsize windows.

`set-transient-gravity` *gravity* [Function]
Set the default gravity for transient/pop-up windows.

`gravity` *gravity* [Command]
Set a window's gravity within its frame. Gravity controls where the window will appear in a frame if it is smaller that the frame. Possible values are:

center

top

right

bottom

left

top-right

top-left

bottom-right
bottom-left

5.3 Controlling Raise And Map Requests

It is sometimes handy to deny a window's request to be focused. The following variables determine such behavior.

A map request occurs when a new or withdrawn window requests to be mapped for the first time.

A raise request occurs when a client asks the window manager to give an existing window focus.

`*deny-map-request*` [Variable]
A list of window properties that stumpwm should deny matching windows' requests to become mapped for the first time.

`*deny-raise-request*` [Variable]
Exactly the same as **deny-map-request** but for raise requests.

Note that no denial message is displayed if the window is already visible.

`*suppress-deny-messages*` [Variable]
For complete focus on the task at hand, set this to `T` and no raise/map denial messages will be seen.

Some examples follow.

```
;; Deny the firefox window from taking focus when clicked upon.
(push '(:class "gecko") stumpwm:*deny-raise-request*)

;; Deny all map requests
(setf stumpwm:*deny-map-request* t)

;; Deny transient raise requests
(push '(:transient) stumpwm:*deny-map-request*)

;; Deny the all windows in the xterm class from taking focus.
(push '(:class "Xterm") stumpwm:*deny-raise-request*)
```

5.4 Programming With Windows

`define-window-slot` *attr* [Macro]
Create a new window attribute and corresponding get/set functions.

`window-send-string` *string* **&optional** (*window* **(current-window)**) [Function]
Send the string of characters to the current window as if they'd been typed.

`*default-window-name*` [Variable]
The name given to a window that does not supply its own name.

5.5 Rule Based Window Placement

`define-frame-preference` *target-group* **&rest** *frame-rules* [Macro]
Create a rule that matches windows and automatically places them in a specified group and frame. Each frame rule is a lambda list:

```
(frame-number raise lock &key create restore dump-name class instance type role t
```

frame-number
The frame number to send matching windows to

raise When non-nil, raise and focus the window in its frame

lock When this is nil, this rule will only match when the current group matches *target-group*. When non-nil, this rule matches regardless of the group and the window is sent to *target-group*. If *lock* and *raise* are both non-nil, then stumpwm will jump to the specified group and focus the matched window.

create When non-NIL the group is created and eventually restored when the value of create is a group dump filename in *DATA-DIR*. Defaults to NIL.

restore When non-NIL the group is restored even if it already exists. This arg should be set to the dump filename to use for forced restore. Defaults to NIL

class The window's class must match *class*.

instance The window's instance/resource name must match *instance*.

type The window's type must match *type*.

role The window's role must match *role*.

title The window's title must match *title*.

`clear-window-placement-rules` [Function]
Clear all window placement rules.

`remember` *lock title* [Command]
Make a generic placement rule for the current window. Might be too specific/not specific enough!

`forget` [Command]
Forget the window placement rule that matches the current window.

`dump-window-placement-rules` *file* [Command]
Dump *window-placement-rules* to FILE.

`restore-window-placement-rules` *file* [Command]
Restore *window-placement-rules* from FILE.

6 Frames

Frames contain windows. All windows exist within a frame.

Those used to ratpoison will notice that this differs from ratpoison's window pool, where windows and frames are not so tightly connected.

`pull-window-by-number` *n* **&optional** (*group* **(current-group)**) [Command]
Pull window N from another frame into the current frame and focus it.

`hsplit` **&optional** (*ratio* **1/2**) [Command]
Split the current frame into 2 side-by-side frames.

`vsplit` **&optional** (*ratio* **1/2**) [Command]
Split the current frame into 2 frames, one on top of the other.

`remove-split` **&optional** (*group* **(current-group)**) (*frame* [Command]
(tile-group-current-frame group)) Remove the specified frame in the specified group (defaults to current group, current frame). Windows in the frame are migrated to the frame taking up its space.

`only` [Command]
Delete all the frames but the current one and grow it to take up the entire head.

`curframe` [Command]
Display a window indicating which frame is focused.

`fnext` [Command]
Cycle through the frame tree to the next frame.

`sibling` [Command]
Jump to the frame's sibling. If a frame is split into two frames, these two frames are siblings.

`fother` [Command]
Jump to the last frame that had focus.

`fselect` *frame-number* [Command]
Display a number in the corner of each frame and let the user to select a frame by number. If *frame-number* is specified, just jump to that frame.

`resize` *width height* [Command]
Resize the current frame by *width* and *height* pixels

`balance-frames` [Command]
Make frames the same height or width in the current frame's subtree.

`fclear` [Command]
Clear the current frame.

`move-focus` *dir* [Command]
Focus the frame adjacent to the current one in the specified direction. The following are valid directions:

up

down

left

right

`move-window` *dir* [Command]
Just like move-focus except that the current is pulled along.

`next-in-frame` [Command]
Go to the next window in the current frame.

`prev-in-frame` [Command]
Go to the previous window in the current frame.

`other-in-frame` [Command]
Go to the last accessed window in the current frame.

`echo-frame-windows` **&optional** (*fmt* ***window-format***) [Command]
Display a list of all the windows in the current frame.

`exchange-direction` *dir* **&optional** (*win* **(current-window)**) [Command]
Exchange the current window (by default) with the top window of the frame in specified direction. (bound to `C-t x` by default)

up

down

left

right

`*min-frame-width*` [Variable]
The minimum width a frame can be. A frame will not shrink below this width. Splitting will not affect frames if the new frame widths are less than this value.

`*min-frame-height*` [Variable]
The minimum height a frame can be. A frame will not shrink below this height. Splitting will not affect frames if the new frame heights are less than this value.

`*new-frame-action*` [Variable]
When a new frame is created, this variable controls what is put in the new frame. Valid values are

`:empty` The frame is left empty

`:last-window`
The last focused window that is not currently visible is placed in the frame. This is the default.

6.1 Interactively Resizing Frames

There is a mode called `iresize` that lets you interactively resize the current frame. To enter the mode use the `iresize` command or type `C-t r`.

The following keybindings apply to the mode:

`C-p`
`Up`
`k` Shrink the frame vertically.

`C-n`
`Down`
`j` Expand the frame vertically.

`C-f`
`Right`
`l` Expand the frame horizontally.

`C-b`
`Left`
`h` Shrink the frame horizontally.

`C-g`
`ESC` Abort the interactive resize. NOTE: This currently doesn't work.

`RET` Select the highlighted option.

`iresize` [Command]
Start the interactive resize mode. A new keymap specific to resizing the current frame is loaded. Hit `C-g`, `RET`, or `ESC` to exit.

`abort-iresize` [Command]
Exit from the interactive resize mode.

`exit-iresize` [Command]
Exit from the interactive resize mode.

`*resize-increment*` [Variable]
Number of pixels to increment by when interactively resizing frames.

6.2 Frame Dumping

The configuration of frames and groups can be saved and restored using the following commands.

`dump-desktop-to-file` *file* [Command]
Dumps the frames of all groups of all screens to the named file

`dump-group-to-file` *file* [Command]
Dumps the frames of the current group of the current screen to the named file.

`dump-screen-to-file` *file* [Command]
Dumps the frames of all groups of the current screen to the named file

`restore-from-file` *file* [Command]
Restores screen, groups, or frames from named file, depending on file's contents.

`place-existing-windows` [Command]
Re-arrange existing windows according to placement rules.

7 The Mode Line

The mode line is a bar that runs across either the top or bottom of a head and is used to display information. By default the mode line displays the list of windows, similar to the output `C-t w` produces.

Alternatively, external panel applications such as the GNOME panel and KDE's kicker may be used. Simply starting one of these programs is enough to set it as the mode line of the head it would like to be on (if the panel is XRandR aware) or whichever head is available. In order to avoid problems displaying menus, configure your panel application for positioning at the top or bottom of the head rather than relying on **mode-line-position**

The mode line can be turned on and off with the `mode-line` command or the lisp function `stumpwm:toggle-mode-line`. Each head has its own mode line. For example:

```
;; turn on/off the mode line for the current head only.
(stumpwm:toggle-mode-line (stumpwm:current-screen)
                          (stumpwm:current-head))
```

The mode line is updated after every StumpWM command.

To display the window list and the current date on the modeline, one might do the following:

```
(setf stumpwm:*screen-mode-line-format*
      (list "%w | "
            '(:eval (stumpwm:run-shell-command "date" t))))
```

`(stumpwm:run-shell-command "date" t)` runs the command `date` and returns its output as a string.

`mode-line` [Command]

A command to toggle the mode line visibility.

`toggle-mode-line` *screen head* **&optional** (*format* **(quote *screen-mode-line-format*)**) [Function]

Toggle the state of the mode line for the specified screen

`*screen-mode-line-format*` [Variable]

This variable describes what will be displayed on the modeline for each screen. Turn it on with the function TOGGLE-MODE-LINE or the mode-line command.

It is a list where each element may be a string, a symbol, or a list.

For a symbol its value is used.

For a list of the form (:eval FORM) FORM is evaluated and the result is used as a mode line element.

If it is a string the string is printed with the following formatting options:

%h	List the number of the head the mode-line belongs to
%w	List all windows in the current group windows using **window-format**
%W	List all windows on the current head of the current group using **window-format**
%g	List the groups using **group-format**

%n The current group's name

%u Using **window-format**, return a 1 line list of the urgent windows, space seperated.

%v Using **window-format**, return a 1 line list of the windows, space separated. The currently focused window is highlighted with fmt-highlight. Any non-visible windows are colored the *hidden-window-color*.

%d Using **time-modeline-string**, print the time.

A number of modules have been written that extends the possible formatting strings. See their documentation for details.

The following variables control the color, position, and size of the mode line. See Chapter 13 [Colors], page 49, for an explanation of how to set these color variables.

`*mode-line-position*` [Variable]
Specifies where the mode line is displayed. Valid values are :top and :bottom.

`*mode-line-border-width*` *1* [Variable]
nil

`*mode-line-pad-x*` [Variable]
nil

`*mode-line-pad-y*` [Variable]
nil

`*mode-line-background-color*` [Variable]
nil

`*mode-line-foreground-color*` [Variable]
nil

`*mode-line-border-color*` [Variable]
nil

`*mode-line-timeout*` [Variable]
The modeline updates after each command, when a new window appears or an existing one disappears, and on a timer. This variable controls how many seconds elapse between each update. If this variable is changed while the modeline is visible, you must toggle the modeline to update timer.

8 Groups

Groups in StumpWM are more commonly known as *virtual desktops* or *workspaces*. Why not create a new term for it?

`gnew` *name* [Command]
Create a new group with the specified name. The new group becomes the current group. If *name* begins with a dot (".") the group new group will be created in the hidden state. Hidden groups have group numbers less than one and are invisible to from gprev, gnext, and, optionally, groups and vgroups commands.

`gnew-float` *name* [Command]
Create a floating window group with the specified name and switch to it.

`gnewbg` *name* [Command]
Create a new group but do not switch to it.

`gnewbg-float` *name* [Command]
Create a floating window group with the specified name, but do not switch to it.

`gnext` [Command]
Cycle to the next group in the group list.

`gprev` [Command]
Cycle to the previous group in the group list.

`gnext-with-window` [Command]
Cycle to the next group in the group list, taking the current window along.

`gprev-with-window` [Command]
Cycle to the previous group in the group list, taking the current window along.

`gother` [Command]
Go back to the last group.

`gmerge` *from* [Command]
Merge *from* into the current group. *from* is not deleted.

`groups` **&optional** (*fmt* ***group-format***) [Command]
Display the list of groups with their number and name. **group-format** controls the formatting. The optional argument *fmt* can be used to override the default group formatting.

`vgroups` **&optional** *gfmt wfmt* [Command]
Like `groups` but also display the windows in each group. The optional arguments *gfmt* and *wfmt* can be used to override the default group formatting and window formatting, respectively.

`gselect` *to-group* [Command]
Select the first group that starts with *substring*. *substring* can also be a number, in which case `gselect` selects the group with that number.

`gmove` *to-group* [Command]
Move the current window to the specified group.

`gkill` [Command]
Kill the current group. All windows in the current group are migrated to the next group.

`grename` *name* [Command]
Rename the current group.

`grouplist` **&optional** (*fmt* ***group-format***) [Command]
Allow the user to select a group from a list, like windowlist but for groups

8.1 Customizing Groups

`*group-formatters*` [Variable]
An alist of characters and formatter functions. The character can be used as a format character in **group-format**. When the character is encountered in the string, the corresponding function is called with a group as an argument. The functions return value is inserted into the string. If the return value isn't a string it is converted to one using `prin1-to-string`.

`*group-format*` [Variable]
The format string that decides what information will show up in the group listing. The following format options are available:

%n Substitutes the group number translated via *group-number-map*, if there are more windows than *group-number-map* then will use the group-number.

%s The group's status. Similar to a window's status.

%t The group's name.

`current-group` **&optional** (*screen* **(current-screen)**) [Function]
Return the current group for the current screen, unless otherwise specified.

9 Screens

StumpWM handles multiple screens.

`snext` [Command]
: Go to the next screen.

`sprev` [Command]
: Go to the previous screen.

`sother` [Command]
: Go to the last screen.

9.1 External Monitors

StumpWM will attempt to detect external monitors (via `xdpyinfo`) at startup. StumpWM refers to each monitor as a head. Heads are logically contained by screens. In a dual-monitor configuration, there will be one screen with two heads. Non-rectangular layouts are supported (frames will not be created in the 'dead zone'.) And message windows will be displayed on the current head–that is, the head to which the currently focused frame belongs.

In addition, StumpWM listens for XRandR events and re-configures the heads to match the new monitor configuration. Occasionally StumpWM will miss an XRandR event, use `refresh-heads` to synchronize the head configuration.

`refresh-heads` **&optional** (*screen* **(current-screen)**) [Command]
: Refresh screens in case a monitor was connected, but a ConfigureNotify event was snarfed by another program.

9.2 Programming With Screens

`current-screen` [Function]
: Return the current screen.

`screen-current-window` *screen* [Function]
: Return the current window on the specified screen

`current-window` [Function]
: Return the current window on the current screen

`*screen-list*` [Variable]
: The list of screens managed by stumpwm.

10 Interacting With Unix

`run-shell-command` *cmd* **&optional** *collect-output-p* [Command]
: Run the specified shell command. If *collect-output-p* is `T` then run the command synchonously and collect the output. Be careful. If the shell command doesn't return, it will hang StumpWM. In such a case, kill the shell command to resume StumpWM.

`programs-in-path` **&optional** *full-path* (*path* **(split-string (getenv PATH) :)**) [Function]
: Return a list of programs in the path. if *full-path* is *t* then return the full path, otherwise just return the filename. *path* is by default the `PATH` evironment variable but can be specified. It should be a string containing each directory seperated by a colon.

`pathname-is-executable-p` *pathname* [Function]
: Return T if the pathname describes an executable file.

`*shell-program*` [Variable]
: The shell program used by `run-shell-command`.

`getenv` *var* [Function]
: Return the value of the environment variable.

`(setf getenv)` *val var* [Function]
: Set the value of the environment variable, *var* to *val*.

11 Interacting With X11

`set-x-selection` *text* **&optional** (*selection* **primary**) [Function]
Set the X11 selection string to *string*.

`get-x-selection` **&optional** *timeout* (*selection* **primary**) [Function]
Return the x selection no matter what client own it.

12 Miscellaneous Commands

The following is a list of commands that don't really fit in any other section.

`emacs` [Command]
: Start emacs unless it is already running, in which case focus it.

`banish` **&optional** *where* [Command]
: Warp the mouse the lower right corner of the current head.

`ratwarp` *x y* [Command]
: Warp the mouse to the specified location.

`ratrelwarp` *dx dy* [Command]
: Warp the mouse by the specified amount from its current position.

`ratclick` **&optional** (*button* **1**) [Command]
: Simulate a pointer button event at the current pointer location. Note: this function is unlikely to work unless your X server and CLX implementation support XTEST.

`echo-date` [Command]
: Display the date and time.

`eval-line` *cmd* [Command]
: Evaluate the s-expression and display the result(s).

`window-send-string` *string* **&optional** (*window* **(current-window)**) [Command]
: Send the string of characters to the current window as if they'd been typed.

`reload` [Command]
: Reload StumpWM using `asdf`.

`loadrc` [Command]
: Reload the `~/.stumpwmrc` file.

`keyboard-quit` [Command]

`quit` [Command]
: Quit StumpWM.

`restart-hard` [Command]
: Restart stumpwm. This is handy if a new stumpwm executable has been made and you wish to replace the existing process with it.

  Any run-time customizations will be lost after the restart.

`restart-soft` [Command]
: Soft Restart StumpWM. The lisp process isn't restarted. Instead, control jumps to the very beginning of the stumpwm program. This differs from RESTART, which restarts the unix process.

  Since the process isn't restarted, existing customizations remain after the restart.

`getsel` [Command]
Echo the X selection.

`putsel` *string* [Command]
Stuff the string *string* into the X selection.

`command-mode` [Command]
Command mode allows you to type ratpoison commands without needing the `C-t` prefix. Keys not bound in StumpWM will still get sent to the current window. To exit command mode, type `C-g`.

`copy-unhandled-error` [Command]
When an unhandled error occurs, StumpWM restarts and attempts to continue. Unhandled errors should be reported to the mailing list so they can be fixed. Use this command to copy the unhandled error and backtrace to the X11 selection so you can paste in your email when submitting the bug report.

`commands` [Command]
List all available commands.

`lastmsg` [Command]
Display the last message. If the previous command was lastmsg, then continue cycling back through the message history.

`list-window-properties` [Command]
List all the properties of the current window and their values, like xprop.

`run-commands` **&rest** *commands* [Function]
Run each stumpwm command in sequence. This could be used if you're used to ratpoison's rc file and you just want to run commands or don't know lisp very well. One might put the following in one's rc file:

```
(stumpwm:run-commands
  "escape C-z"
  "exec firefox"
  "split")
```

`defcommand` *name* (**&rest** *args*) (**&rest interactive-args**) **&body** *body* [Macro]
Create a command function and store its interactive hints in *command-hash*. The local variable %interactivep% can be used to check if the command was called interactively. If it is non-NIL then it was called from a keybinding or from the colon command.

The NAME argument can be a string, or a list of two symbols. If the latter, the first symbol names the command, and the second indicates the type of group under which this command will be usable. Currently, tile-group and floating-group are the two possible values.

INTERACTIVE-ARGS is a list of the following form: ((TYPE PROMPT) (TYPE PROMPT) ...)

each element in INTERACTIVE-ARGS declares the type and prompt for the command's arguments.

TYPE can be one of the following:

:y-or-n A yes or no question returning T or NIL.

:variable A lisp variable

:function A lisp function

:command A stumpwm command as a string.

:key-seq A key sequence starting from *TOP-MAP*

:window-number
An existing window number

:number An integer number

:string A string

:key A single key chord

:window-name
An existing window's name

:direction A direction symbol. One of :UP :DOWN :LEFT :RIGHT

:gravity A gravity symbol. One of :center :top :right :bottom :left :top-right :top-left :bottom-right :bottom-left

:group An existing group

:frame A frame

:shell A shell command

:rest The rest of the input yes to be parsed.

:module An existing stumpwm module

Note that new argument types can be created with DEFINE-STUMPWM-TYPE.

PROMPT can be string. In this case, if the corresponding argument is missing from an interactive call, stumpwm will use prompt for its value using PROMPT. If PROMPT is missing or nil, then the argument is considered an optional interactive argument and is not prompted for when missing.

Alternatively, instead of specifying nil for PROMPT or leaving it out, an element can just be the argument type.

`define-stumpwm-type` *type* (*input prompt*) **&body** *body* [Macro]

Create a new type that can be used for command arguments. *type* can be any symbol.

When *body* is evaluated *input* is bound to the argument-line. It is passed to `argument-pop`, `argument-pop-rest`, etc. *prompt* is the prompt that should be used when prompting the user for the argument.

```
(define-stumpwm-type :symbol (input prompt)
 (or (find-symbol (string-upcase
     (or (argument-pop input)
                          ;; Whitespace messes up find-symbol.
```

```
        (string-trim " "
          (completing-read (current-screen)
    prompt
    ;; find all symbols in the
    ;;  stumpwm package.
    (let (acc)
      (do-symbols (s (find-package "STUMPWM"))
        (push (string-downcase (symbol-name s)) acc))
      acc)))
                        (throw 'error "Abort.")))
                   "STUMPWM")
     (throw 'error "Symbol not in STUMPWM package")))

(defcommand "symbol" (sym) ((:symbol "Pick a symbol: "))
  (message "~a" (with-output-to-string (s)
          (describe sym s))))
```

This code creates a new type called `:symbol` which finds the symbol in the stumpwm package. The command `symbol` uses it and then describes the symbol.

`run-or-raise` *cmd props* **&optional** (*all-groups* ***run-or-raise-all-groups***) (*all-screens* ***run-or-raise-all-screens***) [Function]

Run the shell command, *cmd*, unless an existing window matches *props*. *props* is a property list with the following keys:

`:class` Match the window's class.

`:instance` Match the window's instance or resource-name.

`:role` Match the window's `WM_WINDOW_ROLE`.

`:title` Match the window's title.

By default, the global **run-or-raise-all-groups** decides whether to search all groups or the current one for a running instance. *all-groups* overrides this default. Similarily for **run-or-raise-all-screens** and *all-screens*.

`run-or-pull` *cmd props* **&optional** (*all-groups* ***run-or-raise-all-groups***) (*all-screens* ***run-or-raise-all-screens***) [Function]

Similar to run-or-raise, but move the matching window to the current frame instead of switching to the window.

`*run-or-raise-all-groups*` [Variable]

When this is `T` the `run-or-raise` function searches all groups for a running instance. Set it to NIL to search only the current group.

`*run-or-raise-all-screens*` [Variable]

When this is `T` the `run-or-raise` function searches all screens for a running instance. Set it to `NIL` to search only the current screen. If **run-or-raise-all-groups** is `NIL` this variable has no effect.

`restarts-menu` *err* [Function]
: Display a menu with the active restarts and let the user pick one. Error is the error being recovered from. If the user aborts the menu, the error is re-signalled.

`with-restarts-menu` **&body** *body* [Macro]
: Execute BODY. If an error occurs allow the user to pick a restart from a menu of possible restarts. If a restart is not chosen, resignal the error.

`*startup-message*` [Variable]
: This is the message StumpWM displays when it starts. Set it to NIL to suppress.

`*suppress-abort-messages*` [Variable]
: Suppress abort message when non-nil.

`*default-package*` [Variable]
: This is the package eval reads and executes in. You might want to set this to `:stumpwm` if you find yourself using a lot of internal stumpwm symbols. Setting this variable anywhere but in your rc file will have no effect.

`defprogram-shortcut` *name* **&key** (*command* **(string-downcase (string name))**) (*props* **(quasiquote (quote (class #S(comma :expr (string-capitalize command) :kind 0))))**) (*map* ***top-map***) (*key* **(kbd (concat H- (subseq command 0 1)))**) (*pullp* **nil**) (*pull-name* **(intern1 (concat (string name) -PULL))**) (*pull-key* **(kbd (concat H-M- (subseq command 0 1)))**) [Macro]
: Define a command and key binding to run or raise a program. If *pullp* is set, also define a command and key binding to run or pull the program.

`*initializing*` [Variable]
: True when starting stumpwm. Use this variable in your rc file to run code that should only be executed once, when stumpwm starts up and loads the rc file.

12.1 Menus

Some commands present the options in a menu. The following are the menu key bindings:

`C-p`
`Up`
`k` Highlight the previous menu option.

`C-n`
`Down`
`j` Highlight the next menu option.

`C-g`
`ESC` Abort the menu.

`RET` Select the highlighted option.

12.2 StumpWM's Data Directory

If you want to store StumpWM data between sessions, the recommended method is to store them in `~/.stumpwm.d/`. StumpWM supplies some functions to make doing this easier.

`*data-dir*` [Variable]
The directory used by stumpwm to store data between sessions.

`data-dir-file` *name* **&optional** *type* [Function]
Return a pathname inside stumpwm's data dir with the specified name and type

`with-data-file` (*s file* **&rest** *keys* **&key** *(if-exists supersede)* **&allow-other-keys**) **&body** *body* [Macro]
Open a file in StumpWM's data directory. keyword arguments are sent directly to OPEN. Note that IF-EXISTS defaults to :supersede, instead of :error.

12.3 Debugging StumpWM

`*debug-level*` [Variable]
Set this variable to a number > 0 to turn on debugging. The greater the number the more debugging output.

`*debug-stream*` [Variable]
This is the stream debugging output is sent to. It defaults to *error-output*. It may be more convenient for you to pipe debugging output directly to a file.

`redirect-all-output` *file* [Function]
Elect to redirect all output to the specified file. For instance, if you want everything to go to ~/.stumpwm.d/debug-output.txt you would do:

```
(redirect-all-output (data-dir-file "debug-output" "txt"))
```

12.4 Timers

StumpWM has a timer system similar to that of *Emacs*.

`run-with-timer` *secs repeat function* **&rest** *args* [Function]
Perform an action after a delay of SECS seconds. Repeat the action every REPEAT seconds, if repeat is non-nil. SECS and REPEAT may be reals. The action is to call FUNCTION with arguments ARGS.

`cancel-timer` *timer* [Function]
Remove TIMER from the list of active timers.

`timer-p` *timer* [Function]
Return T if TIMER is a timer structure.

12.5 Getting Help

`describe-key` *keys* [Command]
: Either interactively type the key sequence or supply it as text. This command prints the command bound to the specified key sequence.

`describe-variable` *var* [Command]
: Print the online help associated with the specified variable.

`describe-function` *fn* [Command]
: Print the online help associated with the specified function.

`where-is` *cmd* [Command]
: Print the key sequences bound to the specified command.

`modifiers` [Command]
: List the modifiers stumpwm recognizes and what MOD-X it thinks they're on.

13 Colors

When specifying a color, it is possible to use its X11 Color Name (usually in the file `/etc/X11/rgb.txt`). You can also use a six digit hex string prefixed by a '#' character in the same way that you can specify colors in HTML.

All text printed by stumpwm is run through a coloring engine before being displayed. All color commands start with a '`^`' (caret) character and apply to all text after it.

`^0-9` A caret followed by a single digit number changes the foreground color to the specified color. A '`*`' can be used to specify the normal color. See the color listing below.

`^0-90-9` A caret followed by two digits sets the foreground and background color. The first digit refers to the foreground color and the second digit to the background color. A '`*`' can be used in place of either digit to specify the normal color. See the color listing below.

`^B` Turn on bright colors.

`^b` Turn off bright colors.

`^n` Use the normal background and foreground color.

`^R` Reverse the foreground and background colors.

`^r` Turn off reverse colors.

`^[` Push the current colors onto the color stack. The current colors remain unchanged.

`^]` Pop the colors off the color stack.

`^>` Align the rest of the line to the right of the window.

`^f<n>` Sets the current font to the font at index n in the screen's font list.

`^(<modifier> &rest arguments)`
Allows for more complicated color settings: `<modifier>` can be one of :fg, :bg, :reverse, :bright, :push, :pop, :font and `:>`. The arguments for each modifier differ:

- :fg and :bg take a color as an argument, which can either be a numeric index into the color map or a hexadecimal color in the form of `"#fff"` or `"#ffffff"`.
- :reverse and :bright take either t or nil as an argument. T enables the setting and nil disables it.
- :push and :pop take no arguments. :push pushes the current settings onto the color stack, leaving the current settings intact. :pop pops color settings off the stack, updating the current settings.
- :font takes an integer that represents an index into the screen's list of fonts, or, possibly, a literal font object that can immediately be used. In a string you'll probably only want to specify an integer.
- `:>` takes no arguments. It triggers right-alignment for the rest of the line.

^^ Print a regular caret.

The default colors are made to resemble the 16 VGA colors and are:

0 black

1 red

2 green

3 yellow

4 blue

5 magenta

6 cyan

7 white

There are only 8 colors by default but 10 available digits. The last two digits are left up to the user. Section 13.1 [Behind The Scenes Look At Colors], page 50, for information on customizing colors.

13.1 Behind The Scenes Look At Colors

Color indexes are stored in **colors** as a list. The default list of colors leave 2 slots for the user to choose. If you'd like to use '`Papaya Whip`' and '`Dark Golden Rod 3`' you might eval the following:

```
(setf *colors* (append *colors*
                       (list "PapayaWhip"
                             "DarkGoldenRod3")))
(update-color-map (current-screen))
```

Of course, you can change all the colors if you like.

`parse-color-string` *string* [Function]
Parse a color-coded string into a list of strings and color modifiers

`uncolorify` *string* [Function]
Remove any color markup in STRING

`*colors*` [Variable]
Eight colors by default. You can redefine these to whatever you like and then call (update-color-map).

`update-color-map` *screen* [Function]
Read *colors* and cache their pixel colors for use when rendering colored text.

14 Hooks

StumpWM exports a number of hooks you can use to add customizations; like hooks in Emacs, you add to a hook with the `add-hook` function. for example:

```
(stumpwm:add-hook 'stumpwm:*new-window-hook* 'my-new-window-custos)
```

adds your `my-new-window-custos` function to the list of functions called when a new window appears.

`add-hook` *hook fn* [Macro]

Add *function* to the hook *hook-variable*. For example, to display a message whenever you switch frames:

```
(defun my-rad-fn (to-frame from-frame)
  (stumpwm:message "Mustard!"))

(stumpmwm:add-hook stumpwm:*focus-frame-hook* 'my-rad-fn)
```

`remove-hook` *hook fn* [Macro]

Remove the specified function from the hook.

The following hooks are available:

`*new-window-hook*` [Hook]

A hook called whenever a window is added to the window list. This includes a genuinely new window as well as bringing a withdrawn window back into the window list.

`*destroy-window-hook*` [Hook]

A hook called whenever a window is destroyed or withdrawn.

`*focus-window-hook*` [Hook]

A hook called when a window is given focus. It is called with 2 arguments: the current window and the last window (could be nil).

`*place-window-hook*` [Hook]

A hook called whenever a window is placed by rule. Arguments are window group and frame

`*start-hook*` [Hook]

A hook called when stumpwm starts.

`*internal-loop-hook*` [Hook]

A hook called inside stumpwm's inner loop.

`*focus-frame-hook*` [Hook]

A hook called when a frame is given focus. The hook functions are called with 2 arguments: the current frame and the last frame.

`*new-frame-hook*` [Hook]

A hook called when a new frame is created. the hook is called with the frame as an argument.

`*message-hook*` [Hook]

A hook called whenever stumpwm displays a message. The hook function is passed any number of arguments. Each argument is a line of text.

`*top-level-error-hook*` [Hook]

Called when a top level error occurs. Note that this hook is run before the error is dealt with according to *top-level-error-action*.

`*focus-group-hook*` [Hook]

A hook called whenever stumpwm switches groups. It is called with 2 arguments: the current group and the last group.

`*key-press-hook*` [Hook]

A hook called whenever a key under *top-map* is pressed. It is called with 3 argument: the key, the (possibly incomplete) key sequence it is a part of, and command value bound to the key.

`*root-click-hook*` [Hook]

A hook called whenever there is a mouse click on the root window. Called with 4 arguments, the screen containing the root window, the button clicked, and the x and y of the pointer.

`*mode-line-click-hook*` [Hook]

Called whenever the mode-line is clicked. It is called with 4 arguments, the mode-line, the button clicked, and the x and y of the pointer.

`*urgent-window-hook*` [Hook]

A hook called whenever a window sets the property indicating that it demands the user's attention

`*event-processing-hook*` [Hook]

A hook called inside stumpwm's inner loop, before the default event processing takes place. This hook is run inside (with-event-queue ...).

15 Modules

A module is a ASDF system that adds additional functionality to StumpWM. StumpWM searches for modules in the **data-dir*/modules* directory. By default this is `~/.stumpwm.d/modules`.

Officially supported modules exist in a separate repository within the StumpWM organization on github. You can install the latest copy by issuing `make install-modules` from StumpWM's root source directory. This will run:

```
git clone git@github.com:stumpwm/stumpwm-contrib.git ~/.stumpwm.d/modules
```

`load-module` *name* [Command]
Loads the contributed module with the given NAME.

`list-modules` [Function]
Return a list of the available modules.

`*load-path*` [Variable]
A list of paths in which modules can be found, by default it is populated by any asdf systems found in '*module-dir*' set from the configure script when StumpWM was built, or later by the user using 'add-to-load-path'

`add-to-load-path` *path* [Command]
If 'PATH' is not in '*LOAD-PATH*' add it, check if 'PATH' contains an asdf system, and if so add it to the central registry

`init-load-path` *path* [Function]
Recursively builds a list of paths that contain modules. This is called each time StumpWM starts with the argument '*module-dir'

`find-module` *name* [Function]
nil

15.1 Writing Modules

Make sure to read the Chapter 16 [Hacking], page 55. If you are familiar with writing ASDF then you can jump in and get started. In either case, quicklisp ships a `quickproject` package that makes setting up a new module very easy. After installing quicklisp (see the README.md for a link):

We're going to put our new module in the `modules/` directory of **data-dir** so that it will be immediately loadable by stumpwm.

First make the directory `new-module`, then from a REPL issue:

```
(ql:quickload "quickproject")
(quickproject:make-project #p"~/.stumpwm.d/modules/new-module" :depends-on '(stumpwm)
```

This will create:

```
-rw-rw-r--  1 dave dave   68 Apr  6 19:38 package.lisp
-rw-rw-r--  1 dave dave   53 Mar 16  2014 README.txt
-rw-rw-r--  1 dave dave  271 Mar 16  2014 new-module.asd
```

```
-rw-rw-r--  1 dave dave 1.8K Apr  6 17:51 new-module.lisp
```

The file `new-module.lisp` will contain the actual implementation of your module. ASDF requires two other files in order to understand how to load and compile your module. They are `new-module.asd` and `package.lisp`. In our example, `new-module.asd` should contain:

```
(asdf:defsystem #:new-module
  :serial t
  :description "Describe new-module here"
  :author "Anne N. O'Nymous"
  :license "GPLv3"
  :depends-on (#:stumpwm)
  :components ((:file "package")
               (:file "new-module"))) ; any other files you make go here
```

The `package.lisp` will contain:

```
(defpackage #:new-module
  (:use #:cl :stumpwm))
```

With these two files defined, and the implementation written in `new-module.lisp`, you should be able to load your module.

Before we load it, we have to add the path to our **load-path** . This can be accomplished by running the following from a REPL:

```
(stumpwm:add-to-load-path "~/.stumpwm.d/modules/new-module")
```

You can also run this interactively with `C-t ;`, which is bound to the `colon` command.

Because we've put our module in a sub-directory of the default **module-dir**, it will automatically get added to the **load-path** the next time StumpWM starts. If you choose to develop your module somewhere else (e.g. `~/quicklisp/local-projects`), then you'll have add

```
(add-to-load-path "~/quicklisp/local-projects/new-module")
```

to your `.stumpwmrc`.

When you've finished writing your module, you can distribute it however you see fit. If it becomes very popular, or you would like the StumpWM devs to maintain it (and they agree), you can have your module merged with the stumpwm-contrib repository on github, just open a pull request to start the discussion.

16 Hacking

For those of you who have worked on Free Software projects before, this part should probably be fairly intuitive.

16.1 Hacking: General Advice

1. Pay attention to file names and contents. If you're making changes to mode-line related code, don't put it in `core.lisp`. If you're introducing some completely new featureset, consider putting all of the new code in a new file.
2. Does a command need to be user-visible ("interactive") or is it just called by other commands?
   - If it's not going to be user-visible, you can just use the familiar `(defun foo () ...)` syntax.
   - If you want the command to be used interactively, you use StumpWM's `defcommand` syntax, as in the examples below.

     ```
     (defcommand test (foo bar)
        ((:string "How you're going to prompt for variable foo: ")
         (:number "How you want to prompt for variable bar: "))
        "This command is a test"
        (body...))

     (defcommand test2 () ()
        "This is also a test"
        (body...))

     (defcommand title (args) (interactive-args)
        "Doc string"
        (body...))
     ```

     So basically, inside the first set of parentheses after the function name, you specify what (if any) arguments will be passed to the command. The second set of parentheses tells StumpWM how to get those arguments if they're not explicitly passed to the command. For example,

     ```
     ((:string "What do you want to do: "))
     ```

     will read a string from the input the user provides. The quoted text is the prompt the user will see. Of course, if you were to, say, call the command test, as defined above, from another piece of code, it wouldn't give the prompt as long as you fed it arguments.
3. Note that all commands defined using the `defcommand` syntax are available both to be called with `C-t ;` and from within other lisp programs, as though they had been defun-ned (which, in fact, they have).
4. Any code that depends on external libraries or programs that some users might not have installed should be packaged as a module and placed in the `*data-dir*/modules/` directory.

5. Don't be afraid to submit your patches to the StumpWM mailing list! It may not immediately make it into the official git repository, but individual users might find it useful and apply it to their own setup, or might be willing to offer suggestions on how to improve the code.
6. Remember: StumpWM is designed to run on many lisp systems. If you must use code specific to one or the other, at the very least warn people that it only works with one lisp implementation. Better yet, figure out how to do it in the other distribution and write a statement like this:

   ```
   #+clisp
   (your-clisp-code)
   #+sbcl
   (your-sbcl-code)
   ```

   #to wrap the code for each lisp. Of course, the best option is to find a way to use the same code for clisp and SBCL.

16.2 Hacking: Using git with StumpWM

For quite a while now, StumpWM has been using the git version control system for development. If you're one using one of the official releases, you can get the bleeding-edge source code from the official git repository with a single command:

```
$ git clone git@github.com:stumpwm/stumpwm.git
```

After this, you'll have a complete git repository, along with the complete revision history since the switch. Feel free to play around; git has some important features that actually make this safe!

Before we get to that stuff, though, you're going to want to tell git about yourself so that your information is included in your commits and patches. The very minimum you're going to want to do is:

```
$ git config --global user.name "Anne N. O'Nymous"
$ git config --global user.email "anonymous@foo.org"
```

Be sure to check out the manual for `git-config`–there are several options you might want to set, such as enabling colorized output or changing the editor and pager you use when making commits and viewing logs.

For the sake of argument, let's say you want to make some major changes to both `user.lisp` and `core.lisp`, add a file called `DANGEROUS_EXPERIMENT_DO_NOT_USE_OR_ELSE.lisp`, and remove the manual because you're too 1337 for such things. However, you don't want to break your entire StumpWM setup and start over. Thankfully, you don't have to. Before you get started, issue this command from the stumpwm directory:

```
$ git checkout -b experimental
```

You should now find yourself in a new branch, called experimental. To confirm this, type `git branch`; there should be an asterisk next to the branch you're currently viewing. At any time, you can type `git checkout master` to return to your master branch, and at any time you can have as many branches of the project as you like. If you want to create a new branch based not on the master branch but on your experimental branch, for example, you'd type:

```
$ git checkout -b new-experiment experimental
```

This will place you in a newly-created branch called "new-experiment" which should be identical to your experimental branch as of the last commit (more on that soon). If you're actually typing out the directions, switch back to your old experimental branch like so:

```
$ git checkout experimental
```

Anyway, now that you have a new branch, create that new file with the long name, which I'll just call `danger.lisp` for brevity. Make whatever changes you want to it, and when you're done, tell git about your new file.

```
$ git add dangerous.lisp
```

Now, let's pretend you're done making changes. Tell git you're done for now:

```
$ git commit -a
```

This will open up a prompt in your editor of choice for you to describe your changes. Try to keep the first line short, and then add more explanation underneath (for an example, run the command `git log` and take a look at some of the longer commit explanations). Save that file and then do this:

```
$ git checkout master
$ ls
```

Then look for your new file. It's not there! That's because you've done all of your work in another branch, which git is currently hiding from you so that you can "check out" the branch called "master." All is as it should be–your master repository is still safe.

```
$ git checkout experimental
```

Now, delete `manual.lisp` and `stumpwm.texi`. That's right. Wipe them off the face of the Earth, or at least off the hard drive of your computer. When you're done, you don't have to tell git you've deleted them; it'll figure it out on its own (though things may not compile properly unless you edit `Makefile.in` and `stumpwm.asd`. Anyway, go ahead and edit `core.lisp` and `user.lisp`. Really break 'em. Run free! When you're done, do another commit, as above, and give it a stupid title like "lolz i b0rked stUmpwm guys wTF!?!?!!111!" Now try to compile. Just try. It won't work. If it does, you're some kind of savant or something. Keep up the good work. If you've actually managed to break StumpWM like you were supposed to, never fear! You have two options at this point.

One is to go back to the master branch (with another git checkout) and just delete your experimental branch, like so:

```
$ git branch -D
```

The "`-D`" means to force a delete, even if the changes you've made aren't available elsewhere. A "`-d`" means to delete the branch if and only if you've merged the changes in elsewhere.

The other option is to create patches for each of your commits so far, delete the branch, and then apply any working/wanted patches in a new branch. Create your patches (after committing) like so:

```
$ git format-patch -o patches origin
```

(Before doing that you can review your changes with `git log origin..`)

You can also use the `format-patch` command to create a patch of working code to send in to the mailing list.

A developer might ask you to try out something they're working on. To fetch their master branch, you'd do this:

```
$ git remote add -f -m master -t master foo git://bar.org/~foo/stumpwm
```

Here, "foo" is the shorthand name you'll use to refer to that repository in the future. To checkout a local copy of that repository, you'd then do

```
$ git checkout --track -b foo-master foo/master
```

Later you could use `git pull foo` to update while looking at that branch (and note that `git pull` with no arguments, in the master branch, will update your StumpWM from the official repository).

Finally, if you want to move your experimental changes into your master branch, you'd checkout your master branch and run:

```
$ git merge experimental
```

If there are file conflicts, `git diff` will show you where they are; you have to fix them by hand. When you're done, do another

```
$ git commit -a
```

to finalize the changes to your master branch. You can then delete your experimental branch. Alternately, you can wait until your changes (assuming you sent them in) make it into the official repository before deleting your experimental branch.

16.3 Sending Patches

While patches are still welcome on the mailing list, StumpWM's development has mostly migrated to github's issue tracker. This means you can open a pull request to submit a patch to StumpWM. The following guidelines apply to pull requests and patches sent to the mailing list.

- Make sure it applies clean to the main git repository
- Ensure that you aren't introducing tabs, extra blank lines, or whitespace at the end of lines.
- Ensure your patch doesn't contain irrelevant indenting or reformatting changes.
- Try to make your patch address a single issue. If your patch changes two unrelated issues, break them into two seperate patches that can stand on their own.
- Don't send intermediate patches. When you're working on a feature you might make several commits to your local repository as you refine it and work out the bugs. When it's polished and ready to ship, send it as one patch! Sometimes it makes sense to send it as multiple patches if each patch contains a discrete feature or bug fix that can stand on its own. If one of your patches changes code that was added or modified in an earlier patch, consider merging them together and sending them as one.

Command and Function Index

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Variable Index