CIS681 – Artificial Intelligence – Emacs and Running SBCL on EE/CIS Machines

1 Getting Started

Before starting to work with lisp copy the file m̃ccoy/.sbclrc into your directory. Your .sbclrc file works like your .login file in that it is automatically loaded into lisp as you start the system up. It initially contains some function definitions you will need. (Honestly, the pp function does not appear to work and you will get some warnings when you load the file – you can ignore the warnings for now.) You may add additional lines in order to customize lisp to your needs.

To get into lisp type, `sbcl` as a unix-shell command. You will get some initialization messages and finally the lisp prompt: $*$. You are now in lisp and ready to go!

2 Loading and Editing Your Lisp Files

To load a file into lisp, type: `(load ‘‘file-name’’)`. You will get a number of messages and warnings, but, if it eventually loads correctly, lisp will respons with:

```
T
*
```

and all functions defined in the file loaded will now be defined for Lisp. Don't worry about the warnings.

You can edit your files with any editor, I recommend emacs (or, even better, xemacs).

2.1 Emacs and Lisp

There are (at least) two options for using emacs for your lisp files. The preferred option involves running lisp from the emacs editor. Emacs has a "lisp mode" which is automatically invoked when a file with a .lisp extension is edited.

Some useful emacs commands:

Hold $\Longrightarrow$	Ctrl	Esc	Ctrl-Esc	Press $\Downarrow$
Move to beginning of	line	sentence	definition	a
Move to end of	line	sentence	definition	e
Delete to end of	line	sentence	expression	k
Move back one	character	word	expression	b
Move forward one	character	word	expression	f
Delete next	character	word	expression	d

To run lisp from emacs: This option is preferred. It allows you to have both lisp and your file on the screen at the same time (often helpful in debugging!), and to have the ability to do many emacs commands in the lisp window (such as editing a line, copying a line previously typed, etc.). In addition, emacs will highlight matching parenthesis on your typed input.

First, if you run emacs put the following lines in your .emacs file.

```
;;; The following is for LISP
;;; The SBCL binary and command-line arguments
(setq inferior-lisp-program ''/usr/local/bin/sbcl --noinform'')
```

If you run xemacs, put the following in your .xemacs/init.el file.

```
;;; The following is for LISP
;;; The SBCL binary and command-line arguments
(require 'inf-lisp)
(setq inferior-lisp-program ''/usr/local/bin/sbcl --noinform'')
```

Note: the –noinform is optional.

Get into emacs with the file that your lisp functions are in. Split the screen (<Ctrl>-x 2). In one half of the screen, run an inferior lisp process (<Esc>-x inferior-lisp). This will invoke lisp in the emacs window giving you the ability to use lisp "as you normally would" (except the window allows you to do things like scroll in it and execute a number of emacs commands).

The inferior lisp window gives you many commands that will make your life easier. To see what these commands are, ask the window to describe its mode (you can use the help menu for this or type <Esc>-x describe-mode in the window).

Now you are able to go back and forth between editing your file and running lisp by simply going from one window to the other. Don't forget that you must load the file into lisp each time you change it (that is, the two windows are really independent of each other). (But note from the describe-mode documentation that there are ways to have saved-files loaded automatically.)

3 Debugging

NOTE: I have not tested this all in sbcl yet – but maybe....

3.1 Finding out about Functions

A simple facility is provided for commenting functions for the purpose of on-line documentation. The documentation for each function should be enclosed in double quotes (" ").

Several functions are available for finding out about functions in lisp.

`(grindef 'fn-name)` – prints the definition of the function.

`(describe 'obj-name)` – prints a description of the object (using the documentation lines).

3.2 Debugging Tools

`(trace fn-list)` and `(untrace fn-list)` – Invoking trace with one or more function names (symbols) causes the functions named to be traced. Henceforth, whenever such a function is invoked, information about the call, the arguments passed, and the eventually returned values, if any, will be printed. Calling trace with no arguments will return a list of functions currently being traced.

Invoking untrace with one or more function names will cause those functions not to be traced any more. Calling untrace with no arguments will cause all currently traced functions to be no longer traced.

Consider the following recursive function defined in lisp:

```
(defun fact (n)
  "computes n*(n-1)*...*1"
  (cond ((equal n 1) n)
        ((* n (fact (1- n))))))
```

Once the file is loaded into lisp, we can trace it and watch its execution. Each recursive call will be shown with the arguments, embedded calls will be indented, and the results of the calls will be shown indented evenly with the initial call.

```
> (trace fact)
(FACT)
> (fact 5)
1 Enter FACT 5
| 2 Enter FACT 4
|   3 Enter FACT 3
|   | 4 Enter FACT 2
|   |   5 Enter FACT 1
|   |   5 Exit FACT 1
|   | 4 Exit FACT 2
|   3 Exit FACT 6
| 2 Exit FACT 24
1 Exit FACT 120
120
```

`(step <form>)` – This evaluates form and returns what form returns. However, the user is allowed to interactively "single-step" through the evaluation of form, at least through those evaluation steps that are performed interpretively. Once you get into "step" system help is available.

`(dribble ''filename'')` – Sends a record of the input/output interaction to the named file. The primary purpose of this is to create a readable record of an interactive session. `(dribble)` terminates the recording of input and output and closes the dribble file.

`(shell ''any function call'')` – this function allows you to call any unix command from lisp. Very useful for things like directory listings etc..