Lab class: Background material and introduction to Lisp

Christopher Buckingham
Computer Science
University of Aston

Note: Practical classes will build on and reference previous handouts so please bring them all with you, each week.

1 General background to tutorials and tutorial sheets

The idea behind the tutorials is for you to begin to understand how Lisp works and how to use it for modelling aspects of the world. The tutorial sheets must be read in conjunction with the lecture notes and your understanding of those lectures, otherwise you will not comprehend the context behind the sheets. However, this introductory section summarises the main points you need to know.

1.1 The way to approach Lisp tutorials

Think, experiment, and play. Each tutorial sheet may and probably will require more work than just the one-hour class and I am expecting you to put in additional hours outside lectures and labs. Although there is no separately-assessed assignment, there will be questions on the practical work that you will not be able to answer without having done it yourself.

1.2 Terminology and symbols

When you see a name within brackets like <>, you replace the name *and the brackets* by your own defined name. In other words, the angled brackets are place holders for a particular name. For example, suppose you are asked to write down your name and house number in a specified sentence. The following template describes how the sentence is defined:

```
My name is <your-name> and I live at <your-house-name-or-number>.
```

which might be translated into the following:

```
My name is Christopher and I live at Buckingham Palace.
```

Forgive me if this seems ridiculously obvious and simplistic, but experience shows that every year people continue to type in the angle brackets when following my instructions and examples.

2 Text editor: Emacs

The Emacs text editor is the recommended one for writing your Lisp programs because it lays the code out so that you can easily see the hierarchical relationships. Of course Emacs is also the greatest ever text editor and can itself be programmed using Lisp to extend its functionality. See more at

www.gnu.org/software/emacs/emacs.html

and I have helped you out by printing off the “tour” document. I have also provided a “crib sheet” for Emacs that provides a quick reference for the main commands.

3 Unix

The unix operating system should have been introduced in a previous module of your degree. However, I have also produced some revision/reference handouts that may help you orientate yourself and provide the basics for this module.

4 Emacs and unix

1. Emacs is a text editor for creating lisp programs;
2. Unix is the operating system, in the form of Ubuntu;
3. Both are less familiar to you than Microsoft applications and operating systems.
4. There *is* a learning curve so be patient.
5. Benefits:
   (a) much faster lisp coding once you are familiar with Emacs;
   (b) tools in your arsenal that you simply must know about;
   (c) better CV for job applications.
6. Emacs can be run in a shell (terminal window) or from the drop-down program list.
7. Either way, ensure it puts lisp files in the same directory as the one from where Clisp was launched otherwise Clisp won't find the files.
8. The Emacs `CTRL-XF` command opens a file and shows the directory in the command window at the bottom of Emacs. The way to type this key combination is to hold down the `CTRL` key and type `X` followed by `F` before releasing the `CTRL` key.
   (a) If `CTRL-XF` does not end in the right directory, type the directory path after the last slash;
   (b) then type the file name after the slash;
   (c) use the tab key for file/directory name completion;
   (d) if the file doesn't exist, Emacs will create a new one;
   (e) if it does exist, Emacs will load it.
9. you can see whether the file in Emacs is saved by two asterisks on the bottom status bar (no asterisks mean the file is saved).
10. Commands can be accessed via the menu bar at the top of the window.
11. Note that commands have shortcuts so learn them to save time and be a fully-fledged power user.

5 Unix, Emacs, and Clisp

To get you going with Emacs and Clisp within the linux environment, carry out the following steps:

Task 1

1. Login to Linux.

2. Open up a terminal window (shell).

3. Create a directory with a suitable name for this module using the `mkdir` unix command:

   ```
   mkdir <directory-name>
   ```

4. Move into the directory by the change directory (cd) command:

   ```
   cd <directory-name>
   ```

5. Check you are in the right directory with `pwd` which shows the path to your current directory (PathWay to Directory, pwd).

6. Now run Emacs and Clisp from inside the directory:

   (a) Type `emacs &` in the shell window so that Emacs runs in the background and the shell is still active. If you do not put the ampersand after the command, then Emacs will open as normal but will still have control of the terminal and so anything you type in there will be ignored.

   (b) Type `clisp` in the shell window to bring up the Clisp program. If clisp does not fire up, then you must have forgotten the ampersand for Emacs.

7. Now go into the Emacs window and load in a file called `tutorial1.lisp` by using the Emacs commands `CTRL-XF` followed by the file name, as explained earlier. Note that if the file already existed, you would see its text in the Emacs window but if it does not, then Emacs creates a new file and the window will be blank. However, you should see the file name at the bottom of the Emacs window.

8. Note that if you want to see the files in the current directory of Emacs, type `CTRL-XF .` If you move to one of the file and hit the return key, that file will open in Emacs/.

9. Type into your new file the following and then save the file with `CTRL-XS`.

   ```
   (setf my-house-number 60)
   ```

   The `setf` function takes a symbol as the first argument (`my-house-number`), sets it up as a variable, and assigns the second argument to it as the value.

10. Now go into the clisp interpreter (in the terminal window where you started it, shown by the > cursor symbol.

11. Type `my-house-number` into the interpreter and note what happens:

    The clisp interpreter tries to evaluate the symbol you have typed in by looking for a variable name that points to the value. But you have not defined the variable in Clisp because it was typed into Emacs but not loaded into Clisp, so there is no value and the interpreter returns an error.

    Note the `Break` word before the cursor. This means you are in **debug** mode for clisp. There are lots of new commands available in this mode to find out why the program didn't work but, for now, we want to go back to the ordinary Lisp mode.

12. Type `:Q` or `quit` and see how the cursor goes back to the ordinary state that says you are no longer in debug mode.

    Please make sure that you are not in debug mode when trying to run your programs because they will not always behave as expected if you are. If you see the `Break` *word and you did not want to debug your program, then get back to ordinary mode.*

13. Clisp does not know the symbol value because you did not declare it in the interpreter. Instead, you wrote it within Emacs. For the interpeter to know it, you must load the lisp file into the interpreter. Type into the clisp interpreter `(load "tutorial1.lisp")`

    Be careful to have closing speech marks. Speech marks define a string and if you leave off the closing ones, the lisp interpreter will wait for ever as you keep on typing, thinking that what you are typing is part of the string.

    If the interpreter is behaving strangely, then type `CTRL-C` *to break out of what it is doing. This will put you into debug mode, which you then leave with* `:Q`.

14. Now type `my-house-number` and the interpeter should return 60 because it has loaded in the variable declaration.

15. Now quit lisp by typing `(quit)` or `CTRL-D`. Note that this quit command is contained in parentheses, which differentiates it from the command that gets you out from the debug mode.

16. Change directory back to the home directory using `cd` without any directory name. This takes you back to your home or root directory for your file space. Check you are at the home directory with `pwd`.

17. Start clisp in the new directory.

18. Now try to load in the `tutorial1.lisp` file you are editing in Emacs and see what happens.

    The file name can't be found because clisp is not running in the same directory where your lisp file is located.

    Note The key thing to remember is that clisp looks for files in the same directory from where it was executed. Therefore you should make sure that Emacs saves files in the directory that you were in when clisp was started, as explained above.

In general:

- Create Lisp programs within Emacs with the `".lisp"` extension so that Emacs goes into Lisp editing mode.
- Note that if you try to load a file into Emacs that doesn't exist, Emacs simply creates it as a new one. So there is no difference between loading existing files and new ones in terms of functionality.
- Load Lisp files that you have created in the editor into the Lisp interpreter with the following command (noting what I said earlier about file name replacement for angled brackets):

```
> (load "<filename>.lisp")
```

5.1 Exit from Lisp

Type `(quit)` into the interpreter or `CTRL-D`.

5.2 Programming errors

If an error occurs in the lisp program, you are automatically put into the debugger mode. For example, suppose you type in a variable name that is not known to the interpretor and so does not have a value. This will cause an error as follows:

```
>taylor

*** - EVAL: variable TAYLOR has no value
```

The numerous debug options don't concern us for the moment. What you should notice, though, is that the prompt has changed and shows that you are in the debugger mode because it has `Break` in front of the number. Just return to the normal mode by typing `quit` or `:Q` and you will get back to the top-level prompt (a number in square brackets followed by >.

```
[3]>
```

6 Tasks for getting used to the Lisp environment

6.1 Lisp is interpreted

- Remember that Lisp is interpreted and that you run it within Clisp, which has been invoked in a terminal window from the command line. You can type your Lisp commands directly into Clisp if you wish, but if you then leave Clisp, the commands are all lost. So the normal approach is to write your Lisp program in a file using Emacs and then loading the file into Clisp so that all of its Lisp contents are in the interpreter's memory. This is how you did it to load in the variable definition for `my-house-number`, earlier.
- Whenever you change any Lisp code in the file, the changes will *only be incorporated within Clisp if you reload the file*. If you do not reload the file, then Clisp will be exactly as it was since the last load (plus any additional commands you might have typed in directly to Clisp).
- Clearly you will be reloading the file numerous times as you add to and amend your program.
- You can obtain previous commands in Clisp by using the UP ARROW key. However, if loading the file was done a very long time back, in terms of the number of subsequent commands to Clisp, then you might have to retype the load command again.
- It is sensible to define a fuction that makes it much easier to reload your lisp program. For example, if the file was called "my-very-first-lisp-program.lisp" then typing `(load "my-very-first-lisp-program.lisp")` is pretty painful. Instead, you could define a function that does exactly the same command but can be called as follows, for example: `(myprog)`. The next tasks will show how to do this.

Task 2

1. Move to the directory where you put your `tutorial1.lisp` Lisp file created by Emacs and start Clisp.
2. In Emacs, go into the `tutorial1.lisp` file and on the first line, define a function called *tut* that loads in the file name:

   ```
   (defun tut () (load "tutorial1.lisp"))
   ```

As always, the first symbol after the bracket is a function call. The `defun` function **def**ines a **fun**ction. It expects the first argument to be a symbol that will be the name of the function. The second argument is a list of the parameters the new function will take. The third argument is a list of arbitrary complexity that defines what the function does.

Note that in this case, the `tut` function has no parameters, which is why the brackets following the function name have nothing between them. You could have written `nil` instead of the brackets because both represent the empty list.

3. Now go to Clisp and type `(tut)` in the command line, noting what happens.

   You have gone into debug mode because the new function definition you wrote in Emacs was not loaded into Clisp. Load it in, as explained earlier, using the full file name not your short function call.

4. Now type `(tut)` and you should see the file being loaded in again, this time with your very much more convenient and shorter function call.

   You can use the short function call for loading in your long file names in the same way for any file, of course, but you ALWAYS have to use the original `(load "<filename>")` *the first time Clisp is started. Clisp does not know the short function call until it is defined when the file is first loaded.*

6.2 In general, set up short load commands

Put the command on the first line of your lisp files:

```
(defun <short-name> nil (load "<filename>.lisp"))
```

and remember they only work after the file has been first loaded.

7 Debugging programs

If you need to trace the function calls during execution of a program, you can use the `trace` built-in function:

```
(trace <function-name>)
```

- This will trace the function calls for that named function.
- You can have many functions being traced at any one time.
- If you wish to turn off the trace, type `(untrace)`.
- If you want to turn off just one of the functions, type `(untrace <function-name>)`
- If you want to know which functions are currently being traced, type `(trace)` with no argument.

If a function fails during execution, you will be left in "debug" mode. This can happen if there is any run-time error. For example, suppose you call a function that does not exist:

```
>(no-such-function parameter)

Error: The function NO-SUCH-FUNCTION is undefined.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>
```

- This is the debug mode which allows you to carry out a number of actions to view the execution history of the function and the changing values of variables.
- `:h` will tell you the different debug commands.
- In general, it is probably best not to worry about the debug mode too much. Use it to find out where the program crashed and then review the code for that particular function. You can also try tracing the functions to see how the parameters are changing.
- To return from debug mode, type `:q`, which will put you back to the top level.
- For help on Lisp, type `(help)` into the interpreter and follow instructions . . . but don't expect too much.